

Ravi Kant Soni, Amuthan Ganeshan
Rajesh RV

Spring: Developing Java Applications for the Enterprise

Learning Path

Leverage the power of Spring MVC, Spring Boot, Spring Cloud, and additional popular web frameworks



Packt>

Spring: Developing Java Applications for the Enterprise

**Leverage the power of Spring MVC,
Spring Boot, Spring
Cloud, and additional popular web
frameworks**

A course in three modules



BIRMINGHAM - MUMBAI

Spring: Developing Java Applications for the Enterprise

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: February 2017

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-755-5

www.packtpub.com

Credits

Authors

Ravi Kant Soni
Amuthan Ganeshan
Rajesh RV

Reviewers

Wilkolek Damian
Jeff Deskins
Miguel Enriquez
Bala Sundarasamy
Mattia Tommasone
Rafał Borowiec
Yogendra Sharma

Content Development Editor

Siddhi Chavan

Graphics

Jason Monteiro

Production Coordinator

Shraddha Falebhai

Preface

The Spring Framework is a cutting-edge framework that provides comprehensive infrastructure support for developing Java applications. The Spring Framework handles the infrastructure so that you can focus on your application. It promotes good programming practice by enabling a POJO-based programming model and also provides a good way to structure your application into layers. The Spring Framework is an ocean with a number of features. The Spring MVC Framework is architected and designed in such a way that every piece of logic and functionality is highly configurable

Microservice is an architecture style and pattern in which complex systems are decomposed into smaller services that work together to form larger business services. Microservices are services that are autonomous, self-contained, and independently deployable. In today's world, many enterprises use microservices as the default standard for building large, service-oriented enterprise applications.

The goal of this course is to enlighten readers with Spring framework. From the very first module, you will be able to develop an application using the Spring Framework. This course further follows a pragmatic approach and guidelines for implementing responsive microservices at scale.

What this learning path covers

Module 1, Learning Spring Application Development, Starting with the architecture of Spring Framework and setting up the key components of the Spring application development environment you will learn the configuration of Spring Container and how to manage Spring beans using XML and annotation. Following this you will explore how to implement the request handling layer using Spring annotated controllers. Other highlights include learning how to build the Java DAO implementation layer by leveraging the Data Access Object design pattern, securing your applications against malicious intruders and exploring the Spring Mail Application Programming interface to send and receive mail.

Module 2, Spring MVC Beginner's Guide, Second Edition, progressively teaches you to configure the Spring development environment, architecture, controllers, libraries, and more before moving on to developing a full web application. It begins with an introduction to the Spring development environment and architecture so you're familiar with the know-hows. From here, we move on to controllers, views, validations, Spring Tag libraries, and more. Finally, we integrate it all together to develop a web application.

Module 3, Spring Microservices, walks you through guidelines to implement responsive microservices at scale. We will then deep dive into Spring Boot, Spring Cloud, Docker, Mesos, and Marathon. Next you will understand how Spring Boot is used to deploy autonomous services, server-less by removing the need to have a heavy-weight application server. Later, you will learn how to go further by deploying your microservices to Docker and manage it with Mesos.

What you need for this learning path

In this course, it is assumed that you have a good understanding of the Java programming language, preferably version 1.6 or later, including the Java basic APIs and syntax. You are also expected to have basic understanding of the JDBC API, relational database, and SQL query language.

To run the example in the Module 2, the following softwares will be required:

Java SE Development

Maven

Apache Tomcat

Spring Tool Suite

For the software requirements of Module 3, please refer to the software list provided in the module 3.

Who this learning path is for

This course is intended for Java developers interested in building enterprise-level applications with Spring Framework. Prior knowledge of Java programming and web development concepts (and a basic knowledge of XML) is expected.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files

6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Spring-Developing-Java-Applications-for-the-Enterprise>

We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Learning Spring Application Development

Chapter 1: Introducing the Spring Framework	1
Introducing Spring	2
Evolution of the Spring Framework	7
Spring Framework Architecture	10
Benefits of the Spring Framework	14
Creating an application in Spring	15
Exercise	29
Summary	29
Chapter 2: Inversion of Control in Spring	31
Understanding Inversion of Control	32
Dependency Injection	46
Bean definition inheritance	63
Autowiring in Spring	67
The bean's scope	72
The Spring bean life cycle	78
Exercise	85
Summary	86
Chapter 3: DAO and JDBC in Spring	87
Overview of database	88
The DAO design pattern	89
JDBC without Spring	90
Spring JDBC packages	98
JDBC with Spring	99
What is JdbcTemplate	105
JDBC batch operation in Spring	113
Calling a stored procedure	117

Exercise	121
Summary	121
Chapter 4: Hibernate with Spring	123
Why Object/Relational Mapping?	124
Introducing ORM, O/RM, and O/R mapping	126
Introducing Hibernate	127
Integrating Hibernate with the Spring Framework	131
Hibernate Query Language	150
Hibernate Criteria Query Language	157
Exercise	165
Summary	165
Chapter 5: Spring Security	167
What is Spring Security?	168
Servlet filters review	169
Security use case	172
Spring Security configuration	172
Securing web application's URL access	174
Logging into web application	177
Users authentication	182
Method-level security	186
Let's get down to business	187
Exercise	200
Summary	200
Chapter 6: Spring Testing	201
Testing using JUnit 4	202
Testing using TestNG	206
Agile software testing	208
Create unit tests of the Spring MVC controller	221
Spring MVC test framework	223
Exercise	230
Summary	230
Chapter 7: Integrating JavaMail and JMS with Spring	233
E-mail support in Spring	234
Spring Java Messaging Service	243
Exercise	253
Summary	253
Appendix A: Solutions to Exercises	255
Chapter 1, Introducing the Spring Framework	255
Chapter 2, Inversion of Control in Spring	256

Chapter 3, DAO and JDBC in Spring	258
Chapter 4, Hibernate with Spring	259
Chapter 5, Spring Security	261
Chapter 6, Spring Testing	262
Chapter 7, Integrating JavaMail and JMS with Spring	264
<u>Appendix B: Setting up the Application Database – Apache Derby</u>	<u>267</u>

Module 1

Learning Spring Application Development

Develop dynamic, feature-rich, and robust Spring-based applications using the Spring framework

1

Introducing the Spring Framework

In this chapter, we'll introduce you to the Spring Framework. We'll also summarize some of the other features of Spring. We'll then discuss the Spring Architecture as well as the benefits of the Spring Framework. We will create your first application in Spring and will look into understanding the packaging structure of the Spring Framework. This chapter serves as a road map to the rest of this book.

The following topics will be covered in this chapter:

- Introducing Spring
- Spring Framework Architecture
- Benefits of the Spring Framework
- Creating a first application in Spring

Spring is an open source framework, which was created by Rod Johnson. He addressed the complexity of enterprise application development and described a simpler, alternative approach in his book *Expert One-on-One J2EE Design and Development*, Wrox.

Spring is now a long-time de-facto standard for Java enterprise software development. The framework was designed with developer productivity in mind, and it makes it easier to work with the existing Java and Java EE APIs. Using Spring, we can develop standalone applications, desktop applications, two-tier applications, web applications, distributed applications, enterprise applications, and so on.

As the title implies, we introduce you to the Spring Framework and then explore Spring's core modules. Upon finishing this chapter, you will be able to build a sample Java application using Spring. If you are already familiar with the Spring Framework, then you might want to skip this chapter and proceed straight to *Chapter 2, Inversion of Control in Spring*.

Introducing Spring

Spring is a lightweight **Inversion of Control (IoC)** and aspect-oriented container framework. Historically, it was created to alleviate the complexity of the then J2EE standard, often giving an alternative model. Any Java EE application can benefit from the Spring Framework in terms of simplicity, loose coupling, and testability.

It remains popular due to its simple approach to building applications. It also offers a consistent programming model for different kinds of technologies, be they for data access or messaging infrastructure. The framework allows developers to target discrete problems and build solutions specifically for them

The Spring Framework provides comprehensive infrastructure support for developing Java EE applications, where the Spring Framework handles the infrastructure and so developers can focus on application development.

Considering a scenario of JDBC application without using the Spring Framework, we have a lot of boilerplate code that needs to be written over and over again to accomplish common tasks. Whereas in Spring JDBC application, which internally uses plain JDBC, the `JdbcTemplate` class eliminates boilerplate code and allows the programmer to just concentrate on application-specific logics development

- For a plain JDBC application without Spring, follow these steps:
 1. Register driver with the `DriverManager` service.
 2. Establish a connection with the database.
 3. Create a statement object.
 4. Prepare and execute an SQL query.
 5. Gather and process the result.
 6. Perform exception handling.
 7. Perform transaction management.
 8. Close JDBC object.

- For a Spring JDBC application (internally uses plain JDBC), follow these steps:
 1. Get access to `JdbcTemplate`.
 2. Prepare and execute an SQL query.
 3. Gather and process the result.

Spring's main aim is to promote good programming practice such as coding to interfaces and make Java EE easier to use. It does this by enabling a **Plain Old Java Object (POJO)**-based programming model, which can be applicable in a wide range of development environments.

Technically, a POJO is any ordinary object that should not implement pre-specified interface or extend pre-specified class or contains annotation

The following is the code for the `POJOClass.java` class:

```
package com.packt.spring.chapter1;

/* This is a simple Java Class - POJO */
public class POJOClass {

    private String message;

    public String getMessage() {
        return this.message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

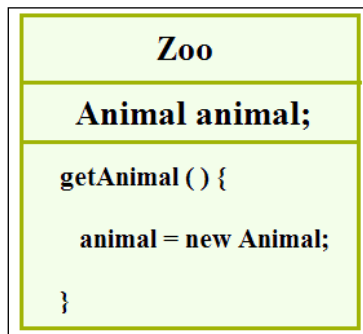
In the preceding code snippet, we have `POJOClass` containing a field and corresponding getter and setter methods. This class is a POJO class as it is not extending or implementing any class or predefined interface of Spring API

Spring is modular, allowing you to use only those parts that you need, without having to bring in extra complexity. The Spring Framework can be used either for all layer implementations or for the development of particular layer of an application.

Features of Spring

The Spring Framework contains the following features:

- **Lightweight:** Spring is described as a lightweight framework when it comes to size and transparency. A lightweight framework helps in reducing complexity in application code. It also helps in avoiding unnecessary complexity in its own functioning. A lightweight framework won't have a high startup time and will run in any environment. A lightweight framework also won't involve huge binary dependencies.
- **Non-intrusive:** This means that your domain logic code has no dependencies on the framework itself. The Spring Framework is designed to be non-intrusive. The object in a Spring-enabled application typically has no dependencies on any predefined interface or class given by Spring API. Thus, Spring can configure application objects that don't import Spring APIs.
- **Inversion of Control (IoC):** Spring's container is a lightweight container that contains Spring beans and manages their life cycle. The core container of the Spring Framework provides an implementation for IoC supporting injection. IoC is an architectural pattern that describes the Dependency Injection needs to be done by external entity rather than creating the dependencies by the component itself. Objects are passively given their dependencies rather than creating dependent objects for themselves. Here, you describe which components need which service, and you don't directly connect your services and components together in your code. Let's consider an example: we have two classes `Zoo` and `Animal`, where `Zoo` has an object of `Animal`:
 - **Without Dependency Injection:** This is a common way to instantiate an object is with a `new` operator. Here, the `Zoo` class contains the object `Animal` that we have instantiated using a `new` operator, as shown in the following screenshot:



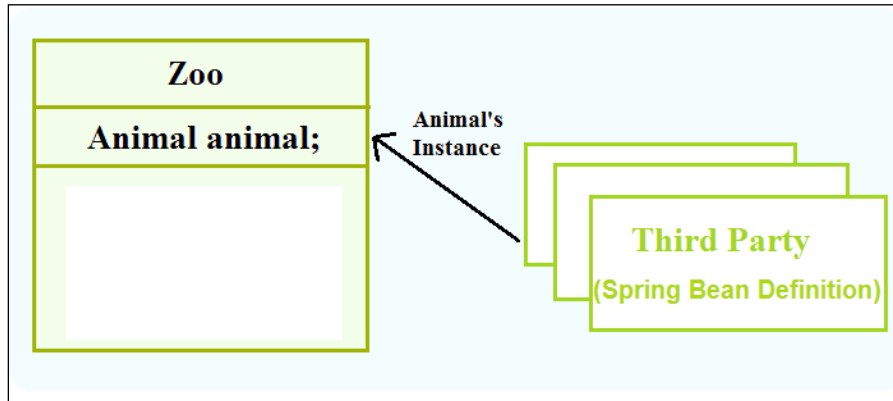
```
    Zoo
    Animal animal;

    getAnimal () {

        animal = new Animal;

    }
```

- **With Dependency Injection:** Here, we supply the job of instantiating to a third party, as shown in following screenshot. `Zoo` needs the object of `Animal` to operate, but it outsources instantiation job to some third party that decides the moment of instantiation and the type to use in order to create the instance. This process of outsourcing instantiation is called dependency injection.

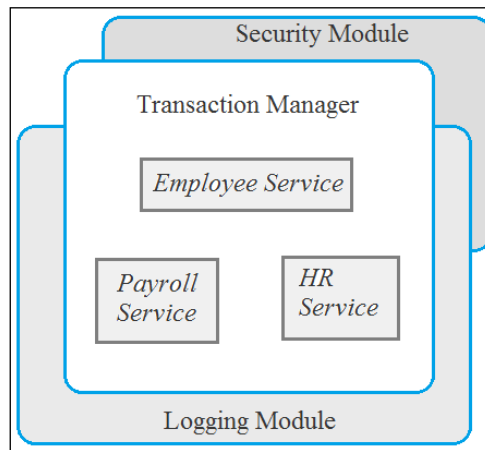


The Spring Framework promotes loose coupling by using the technique known as IoC. We'll talk more about IoC in *Chapter 2, Inversion of Control in Spring*.

- **Aspect-oriented Programming (AOP):** This refers to the programming paradigm that isolates supporting functions from the main program's business logic. It allows a developer to build the core functionality of a system without being aware of additional requirements.

AOP is used in the Spring Framework to provide declarative aspects such as transactions and security. Here, application objects perform business logic and are not responsible for other system concerns such as logging, security, auditing, locking, and event handling. AOP is a method of applying middleware services such as security service, and transaction management service on Spring's application.

Let's consider a payroll management application where there will be **Employee Service**, **HR Service**, and **Payroll Service**, as shown in the following figure, which will perform some functional requirement to the system such as add/update employee details, remove employee, browse employee details, and much more. While implementing business functionality, this type of application would also require nonfunctional capabilities such as role-based access and logging details. AOP leaves an application component to focus on business functionality. Here, the core application implements the business functionality and is covered with layers of functionality provided by AOP for security, logging, and transaction management.



Aspects can be added or removed as needed without changing your code. Spring aspects can be configured using its own IoC container. Spring AOP includes advisors that contain advice and `pointcut` filtering.

- **JDBC exception handling:** The JDBC abstraction layer of the Spring Framework provides an exception hierarchy. It shortens the error handling strategy in JDBC. This is one of the areas where Spring really helps in reducing the amount of boilerplate code we need to write in the exception handling. We'll talk more on Spring JDBC in *Chapter 3, DAO and JDBC in Spring*.
- **Spring MVC Framework:** This helps in building robust and maintainable web applications. It uses IoC that provides separation of controller logic. Spring MVC Framework, which is a part of the Spring Framework licensed under the term of Apache license, is an open source web application framework. Spring MVC Framework offers utility classes to handle some of the most common tasks in web application development.

- **Spring Security:** This provides a declarative security mechanism for Spring-based applications, which is a critical aspect of many applications. We'll add Spring Security to our web applications in *Chapter 7, Spring Security*.

Other features of Spring

The following are the other features provided by the Spring Framework:

- **Spring Web Services:** This provides a contract-first web services model, whereby service implementations are written to satisfy the service contract. For more information, check out <http://static.springsource.org/spring-ws/sites/2.0>.
- **Spring Batch:** This is useful when it's necessary to perform bulk operations on data. For more information, refer to <http://static.springsource.org/spring-batch>.
- **Spring Social:** Social networking, nowadays, is a rising trend on the Internet, and more and more applications such as Facebook and Twitter are being outfitted with integration into social-networking sites. To know more, have a look at <http://www.springsource.org/spring-social>.
- **Spring Mobile:** Mobile applications are another significant area of software development. Spring Mobile supports development of mobile web applications. More information about Spring Mobile can be found at <http://www.springsource.org/spring-mobile>.

Evolution of the Spring Framework

The Spring Framework is an open source framework that has multiple versions released with the latest one being 4.x. The different versions of the Spring Framework are as follows:

- **Spring Framework 1.0:** This version was released on March 2004, and the first release was Spring Framework 1.0 RC4. The final and stable release was Spring Framework 1.0.5. Spring 1.0 was a complete Java/J2EE application framework, which covered the following functionalities:
 - **Spring Core:** This is a lightweight container with various setter and constructor injection
 - **Spring AOP:** This is an **Aspect-oriented Programming (AOP)** interception framework integrated with the core container
 - **Spring Context:** This is an application context concept to provide resource loading

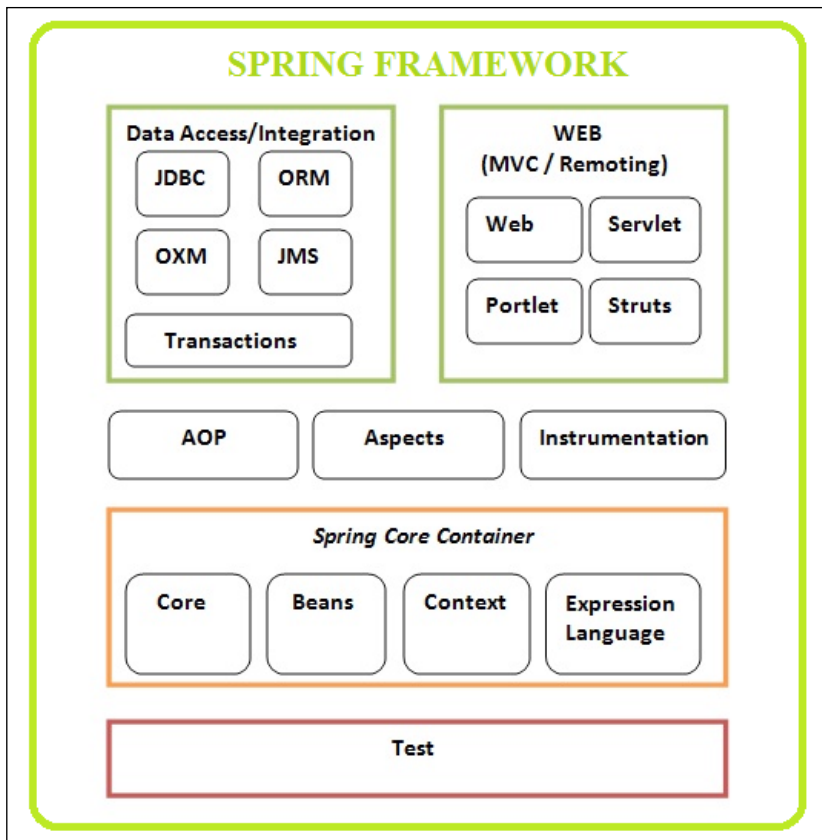
- **Spring DAO:** This is a generic DAO support that provides access to a generic data exception hierarchy with any data access strategy
- **Spring JDBC:** This is a JDBC abstraction shorten error and resource handling
- **Spring ORM:** This is a hibernate support `SessionFactory` management
- **Spring Web:** This web MVC Framework integrates various view technologies
- **Spring Framework 2.X:** The Spring Framework 2.0 was released in October 2006 and Spring 2.5 was released in November 2007. The Spring Framework 2.x release was based around two themes: simplicity and power. This provides you with the following features:
 - Improvements in the IoC container and AOP, including the `@AspectJ` annotation support for AOP development
 - Introduction of bean configuration dialects
 - XML-based configuration is reduced and XML schema support and custom namespace is introduced
 - Annotation-driven configuration that requires component scanning to auto-detect annotated components in the classpath using annotations such as `@Component` or specialized annotations such as `@Repository`, `@Service`, and `@Controller`
 - Introduces annotations such as `@RequestMapping`, `@RequestParam`, and `@ModelAttribute` for MVC controllers
- **Spring Framework 3.0:** This version was released in December 2009. It makes the entire Spring code base to take advantage of the Java 5.0 technology. This provides you with the following features:
 - Supports REST in Spring MVC, which is one of the beautiful additions to the Spring Framework itself.
 - Introduces new annotations `@CookieValue` and `@RequestHeader` for pulling values from cookies and request headers, respectively. It also supports new XML namespace that makes easier to configure Spring MVC.
 - Task scheduling and asynchronous method execution with annotation support is introduced to this version.

- Spring Framework 3.0.5 is the latest update release, which was released on October 20, 2010. The Hibernate version 3.6 final is supported by this Spring release.
- **Spring Framework 3.1:** This version was released in December 2011. This release introduced many new exciting features that are related to cache abstraction, bean definition profiles, environment abstraction, PropertySource abstraction, and a lot more. This provides you with the following features:
 - Introduces Cache Abstraction to add caching concept to any existing application using `@Cacheable` annotation.
 - Introduces annotation called `@Profile`, which is used while applying configuration classes.
 - Introduces PropertySource that is an abstraction performed over any different source of the key-value pairs. In `DefaultEnvironment`, there are two configured PropertySource objects: `System.getProperties()` and `System.getenv()`.
 - Hibernate 4.x is supported by this release through **Java Persistence API (JPA)**. With this release, the `JPA EntityManagerFactory` can be bootstrapped without `persistence.xml` or other metadata files.
 - Introduces `@RequestPart` annotation to provide access to multipart form-data content on the controller method arguments.
 - Introduces the `c:namespace` to support constructor injection.
- **Spring Framework 3.2.x:** This version was released in November 2013. This release introduced the following new features and enhancements to earlier features:
 - Servlet 3-based asynchronous request processing is supported in this release.
 - Supports Java 7 features.
 - Testing of Spring MVC applications without a Servlet container is supported in this release. Here, `DispatcherServlet` is used for server-side REST tests and `RestTemplate` for client-side REST tests.
 - `ContentNegotiationStrategy` is introduced to resolve the requested media types from an incoming request. It also supports Jackson JSON 2 library.

- Method annotated with `@ExceptionHandler`, `@InitBinder`, and `@ModelAttribute` can be added to a class annotated with the `@ControllerAdvice` annotation.
- The `@MatrixVariable` annotation for extracting matrix variables from the request URI is introduced.
- The `@DateTimeFormat` annotation to remove dependency on the Joda-Time library is introduced.
- **Spring Framework 4.x:** This version supports a few new features. Improvements in Spring 4.X include support for Java SE 8, Groovy 2, and a few aspects of Java EE7. This provides you with the following features:
 - Supports external bean configuration using a Groovy DSL
 - Auto-wiring is based on generic types
 - Introduces the `@Description` annotation
 - Introduces `@Conditional` that can conditionally filter the beans.
 - Introduces the `@Jms` annotation to support annotation-driven endpoint
 - Catching support is revisited, provided `CacheResolver` to resolve caches at runtime
 - Added new testing features such as SQL Script execution, bootstrap strategy, and so on
 - Added lightweight messaging and WebSocket-style architectures

Spring Framework Architecture

Spring packaging is modular, allowing you to pick and choose the modules that are applicable to you, without any need to bring in the rest. The following section gives you a detailed explanation about different modules available in the Spring Framework. The following figure shows you a complete overview of the framework and modules supported by the Spring Framework:



Spring Core Container

Spring Core Container consists of the core, beans, context, and expression language modules, as shown in the preceding figure. Let's discuss these in detail as follows

- **Core module:** This module of Spring Core Container is the most important component of the Spring Framework. It provides features such as IoC and Dependency Injection. The idea behind IoC is similar to the Hollywood principle: "Don't call me, I'll call you." Dependency Injection is the basic design principle in Spring Core Container that removes explicit dependence on container APIs.

- **Beans module:** The bean module in Spring Core Container provides `BeanFactory`, which is a generic factory pattern that separates the dependencies such as initialization, creation, and access of the objects from your actual program logic. `BeanFactory` in Spring Core Container supports the following two scopes modes of object:
 - **Singleton:** In singleton, only one shared instance of the object with a particular name will be retrieved on lookup. Spring singleton returns a same bean instance per Spring IoC container. Each time you call `getBean()` on `ApplicationContext`, Spring singleton returns the same bean instance.
 - **Prototype or non-singleton:** In prototype, each retrieval results in the creation of a brand new instance. Each time you call `getBean()` on `ApplicationContext`, Spring prototype creates a separate bean instance.
- **Context module:** An `ApplicationContext` container loads Spring bean definitions and wires them together. The `ApplicationContext` container is the focal point of the Context module. Hierarchical context is also one of the focal points of this API. `ApplicationContext` supports the Message lookup, supporting internationalization (i18N) messages.
- **Expression language: Spring Expression Language (SpEL)** is a powerful expression language supporting the features for querying and manipulating an object graph at runtime. SpEL can be used to inject bean or bean property in another bean. SpEL supports method invocation and retrieval of objects by name from IoC container in Spring.

The AOP module

Spring's **Aspect-oriented Programming (AOP)** module is one of the main paradigms that provide an AOP implementation. Spring AOP module is a proxy-based framework implemented in Java. The Spring Framework uses AOP for providing most of the infrastructure logic in it.

AOP is a mechanism that allows us to introduce new functionalities into an existing code without modifying its design. AOP is used to weave cross-cutting aspects into the code. The Spring Framework uses AOP to provide various enterprise services, such as security in an application. The Spring AOP framework is configured at runtime.

Spring integrates with AspectJ, which is an extension of AOP. AspectJ lets programmers define special constructs called Aspects, which contains several entities unavailable to standard classes.

Data access/integration

Spring's data access addresses common difficulties developers face while working with databases in applications.

- **JDBC module:** The Spring Framework provides solution for various problems identified using JDBC as low-level data access. The JDBC abstraction framework provided under the Spring Framework removes the need to do tedious JDBC-related coding. The central class of Spring JDBC abstraction framework is the `JdbcTemplate` class that includes the most common logic in using the JDBC API to access data such as handling the creation of connection, statement creation, statement execution, and release of resource. The `JdbcTemplate` class resides inside the `org.springframework.jdbc.core` package.
- **ORM module:** The **Object-relational mapping (ORM)** module of the Spring Framework provides a high-level abstraction for ORM APIs, including JPA and Hibernate. Spring ORM module reduces the complexity by avoiding the boilerplate code from application.
- **OXM module:** Spring OXM module stands for Spring Object XML Mappers, which supports Object/XML mapping. It also supports integration with Castor, JAXB, XmlBeans, and the XStream framework.

Most applications need to integrate or provide services to other applications. One common requirement is to exchange data with other systems, either on a regular basis or in real time. In terms of the data format, XML is the most commonly used format. As a result, there exists a common need to transform a `JavaBean` into XML format and vice versa.

Spring supports many common Java-to-XML mapping frameworks and, as usual, eliminates the need for directly coupling to any specific implementation. Spring provides common interfaces for marshalling (transforming `JavaBeans` into XML) and unmarshalling (transforming XML into Java objects) for DI into any Spring beans. Spring also has modules to convert data to and from JSON, in addition to OXM.

- **JMS module:** The **Java Messaging Service (JMS)** module comprises features to produce and consume messages. It is a **Java Message Oriented Middleware (MOM)** API for sending messages between two or more clients. JMS is a specification that describes a common way for Java program to create, send, and read distributed enterprise messages.
 - **Spring Java mail:** The `org.springframework.mail` package is the root package that provides mail support in the Spring Framework. It handles electronic mail.

- **Transaction module:** The Spring transaction module provides abstraction mechanism to supports programmatic and declarative transaction management for classes.

The Web module

The Web module consists of the Web, Servlet, Struts, and Portlet modules.

- **Web module:** The Spring Web module builds on the application context module and includes all the support for developing robust and maintainable web application in a simplified approach. It also supports multipart file-upload functionality.
- **Servlet module:** In Spring, the Servlet module contains **Model-View-Controller (MVC)** implementation that helps to build enterprise web applications. In Spring Framework, the MVC provides clean separation between binding request parameter, business objects, and controller logic.
- **Struts module:** The Web Struts module supports integration of Struts Web tier within a Spring application. It also supports configuration of Struts Actions using Spring Dependency Injection.
- **Portlet module:** Spring Portlet supports for easier development of web application using Spring. Portlet is managed by the Portlet container, similar to the web container. Portlet is used in the UI layer for displaying contents from data source for end user.

The Test module

In the Spring Framework, the Test module helps to test applications developed using the Spring Framework, either using JUnit or TestNG. It also helps in creating mock object to perform unit testing in isolation. It supports running integration tests outside the application server. We'll look at Spring's Test module in *Chapter 6, Spring Testing*.

Benefits of the Spring Framework

The following is the list of a few great benefits of using the Spring Framework

- Spring is a powerful framework, which address many common problems in Java EE. It includes support for managing business objects and exposing their services to presentation tier component.
- It facilitates good programming practice such as programming using interfaces instead of classes. Spring enables developers to develop enterprise applications using POJO and POJI model programming.

- It is modular, allowing you to use only those parts that you need. It allows us to just choose any part of it in isolation.
- It supports both XML- and annotation-based configuration.
- Spring provides a lightweight container that can be activated without using web server or application server software.
- It gives good support for IoC and Dependency Injection results in loose coupling.
- The Spring Framework supports JDBC framework that improves productivity and reduces the error.
- It provides abstraction on ORM software to develop the ORM persistence logic.
- The Spring Web MVC framework provides powerful and a flexible Web framework as an alternative to Struts and other framework.
- The Spring Test module provides support for an easy-to-test code.

Creating an application in Spring

Before we create an application in Spring, first we need to obtain Spring Library. We can download the Spring distribution ZIP files that are available in the Spring Maven Repository. Else, we can simply add the dependencies for Spring into project's `pom.xml` file whenever we use Maven for application development

Spring packaging is modular, allowing you to pick and choose the component you want to use in your application. Spring comes with a large selection of sample applications that can be referred while building your application.

Obtaining Spring JAR files

- Downloading Spring distribution ZIP files: The complete Spring Framework library can be downloaded by opening the link <http://repo.spring.io/release/org/springframework/spring/> and selecting the appropriate subfolder for the version needed for your application development. Distribution ZIP files end with `dist.zip`, for example, `spring-framework-4.1.4.RELEASE-dist.zip`.



While writing this book, the latest version was Spring Framework 4.1.4.

Download the package and extract it. Under the `lib` folder, you will find a list of Spring JAR files that represents each Spring module.

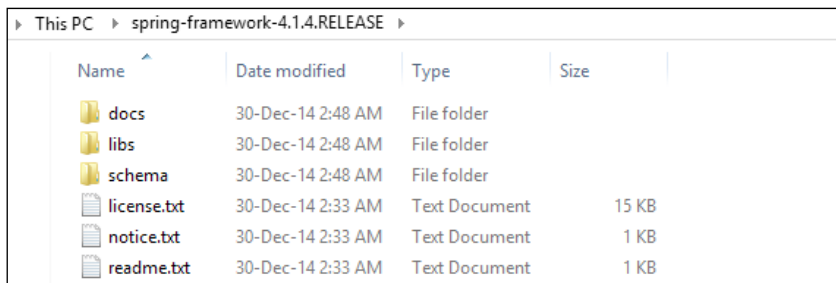
- **Checking Spring out of GitHub:** You can check out the latest version of code from Spring's GitHub repository at <https://github.com/spring-projects/spring-framework>.

To check out the latest version of the Spring code, first install Git from <http://git-scm.com/>, open the Git Bash tool, and run the following command:

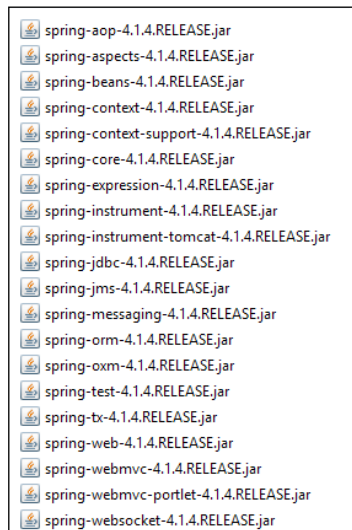
```
git clone https://github.com/spring-projects/spring-framework
```

Understanding Spring packaging

After extracting the downloaded Spring Framework ZIP file, you will get the directory structure, as shown in the following screenshot:



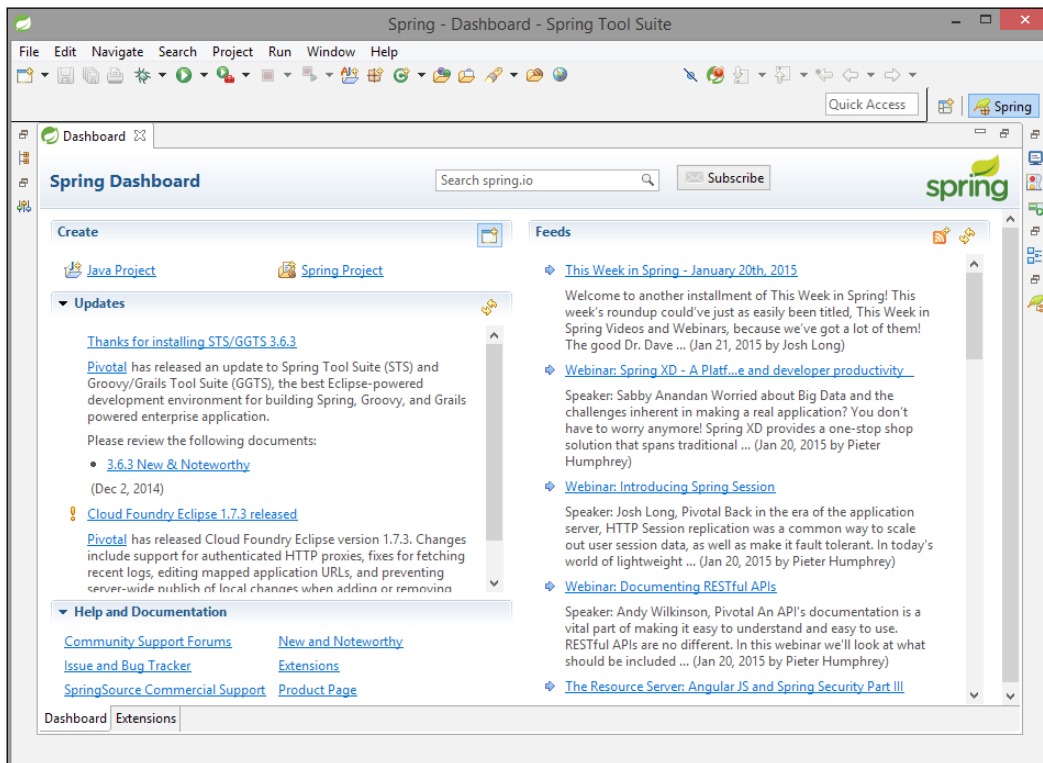
The spring-framework-4.14.RELEASE folder, as shown in the preceding screenshot, contains docs, libs, and schema subfolders. The lib folder contains the Spring JAR files, as shown in the following screenshot



Shown in the preceding screenshot is a list of JAR files required while developing applications using Spring. You can find more details on these JAR file at <http://www.learnr.pro/content/53560-pro-spring/40> and <http://agile-hero.iteye.com/blog/1684338>.

SpringSource Tool Suite

SpringSource Tool Suite (STS) is a powerful Eclipse-based development environment for developing Spring application. The latest version of STS can be downloaded from <http://spring.io/tools/sts>. We will use STS IDE for all our examples in this book. The following screenshot shows a snapshot of an STS dashboard:



Let's now create a simple Spring application using Spring STS.

The Spring application

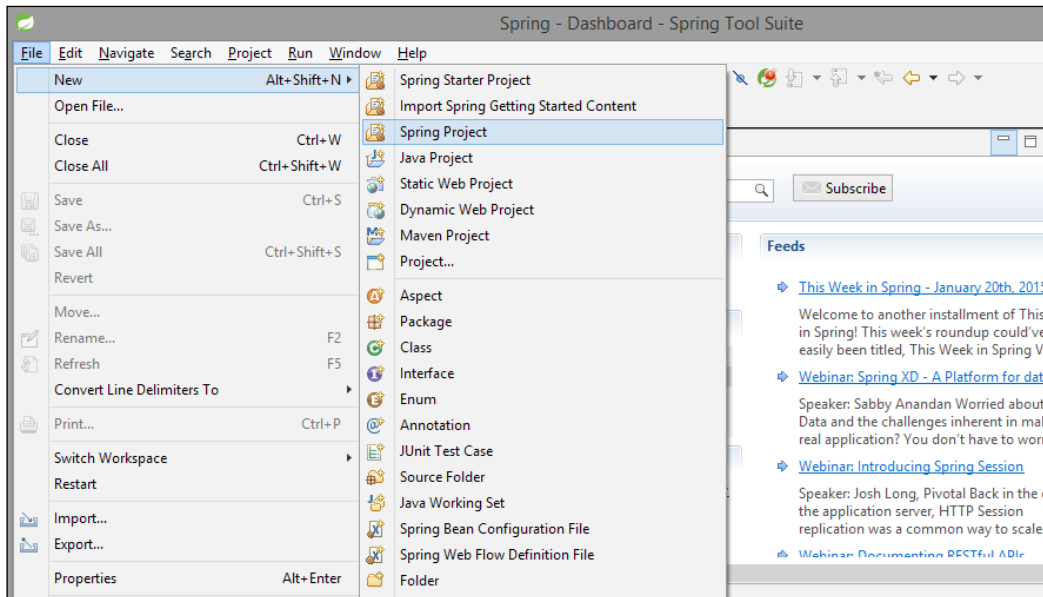
With basic understanding of the Spring Framework, we can now create a simple Spring application example. All the examples in this book have been written using the STS IDE.

We will write a simple Spring application that will print `greeting` message to user. Do not worry if you do not fully understand all the code in this section; we'll go into much more detail on all the topics as we proceed through this book.

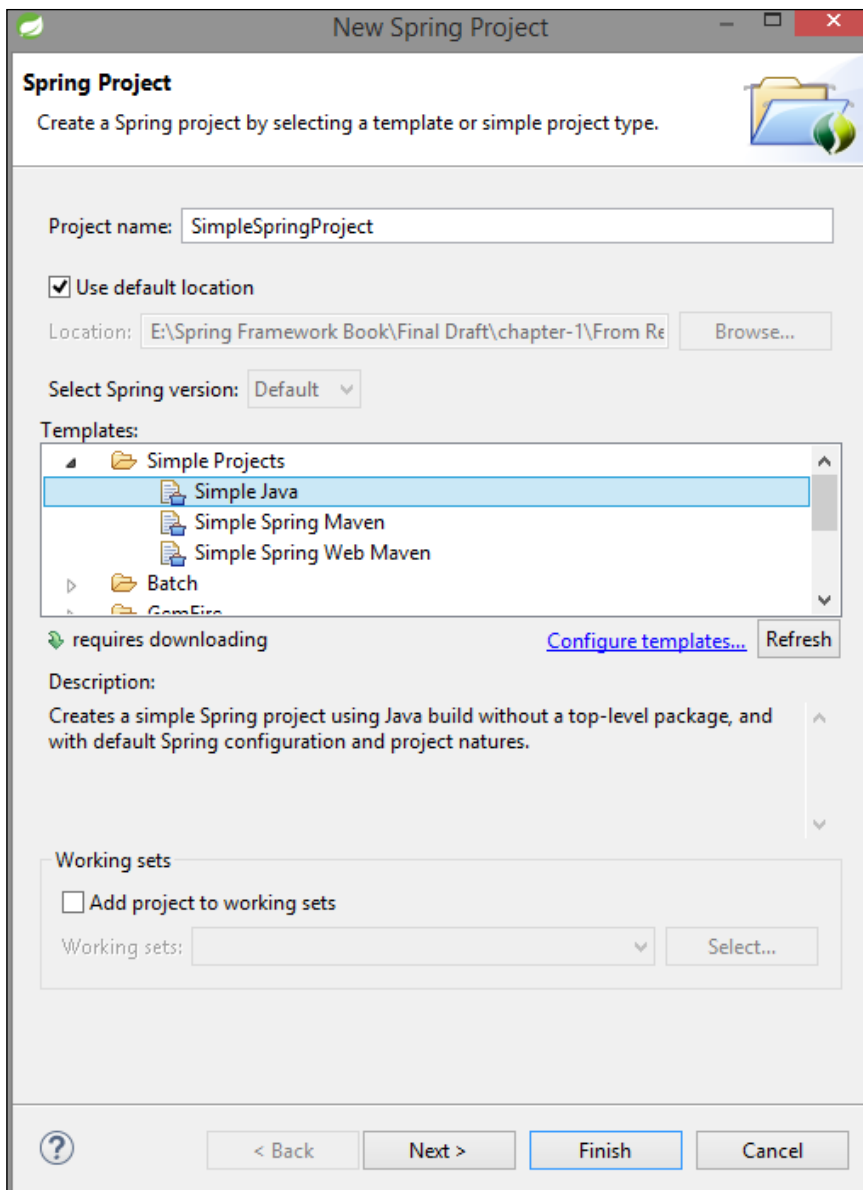
Creating a Spring project

The following steps will help you create your Spring project in STS:

1. The first step in creating a Spring application is to create a new Spring project in the STS IDE. Navigate to **File | New | Spring Project**, as shown in the following screenshot:



2. Name your Spring project `SimpleSpringProject` and select the **Simple Java** template, which creates a Simple Spring project using the Java build without a top-level package and with default Spring configuration and project natures, as shown in the following screenshot:

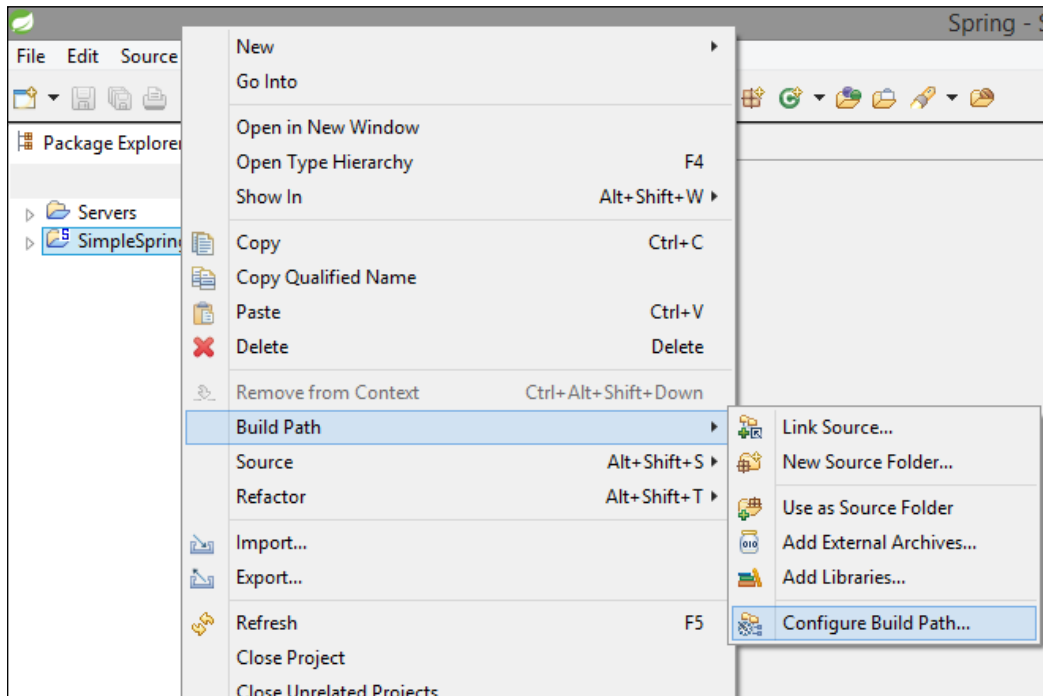


3. Then, click on **Finish**, which will create the project in a workspace.

Adding required libraries

Let's add the basic Spring JAR files to the build path of this Spring project

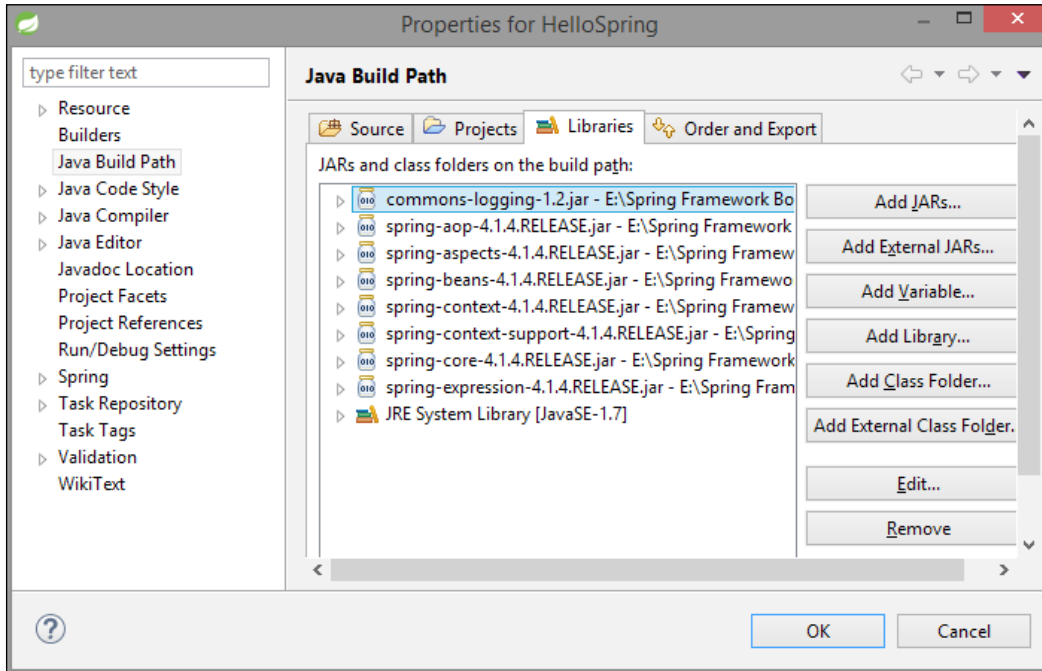
1. Add the Spring Framework libraries and common logging API libraries to your project. The common login library can be downloaded from http://commons.apache.org/proper/commons-logging/download_logging.cgi. To add required libraries, right-click on the project named SimpleSpringProject and then click on the available options in the context menu, that is, **Build Path | Configure Build Path** to display the **Java Build Path** window, as shown in the following screenshot:



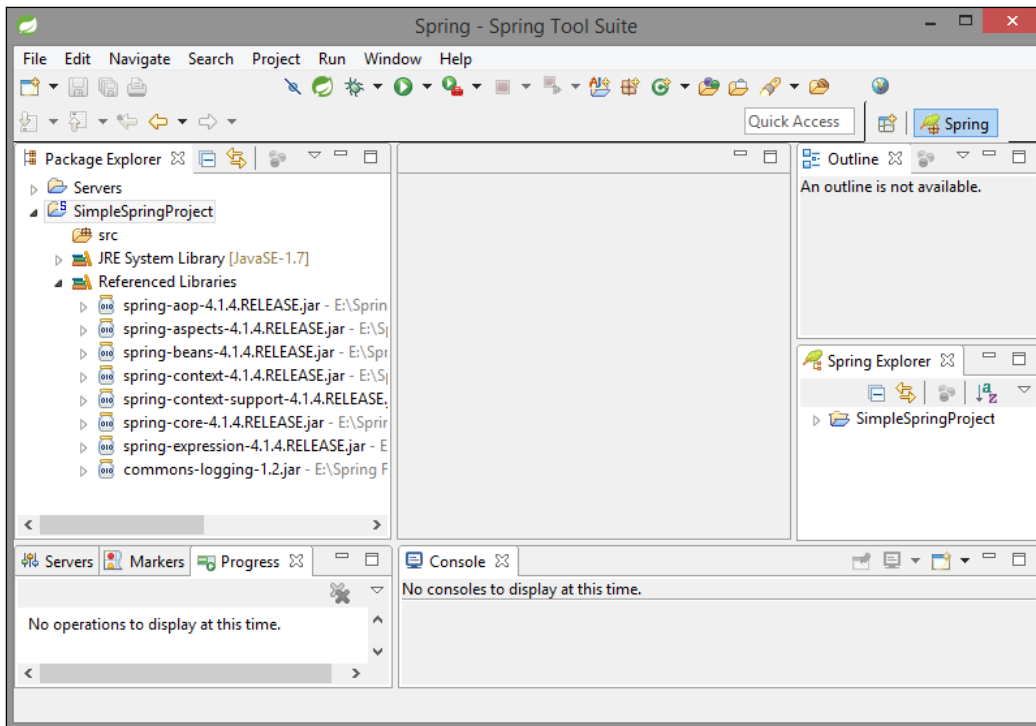
2. Now, use the **Add External JARs** button from the **Libraries** tab in order to include the following core JARs from the Spring Framework and common logging installation directories:
 - spring-aop-4.1.4.RELEASE
 - spring-aspects-4.1.4.RELEASE
 - spring-beans-4.1.4.RELEASE
 - spring-context-4.1.4.RELEASE

- spring-context-support-4.1.4.RELEASE
- spring-core-4.1.4.RELEASE
- spring-expression-4.1.4.RELEASE
- commons-logging-1.2

The **Libraries** tab is as shown in the following screenshot:



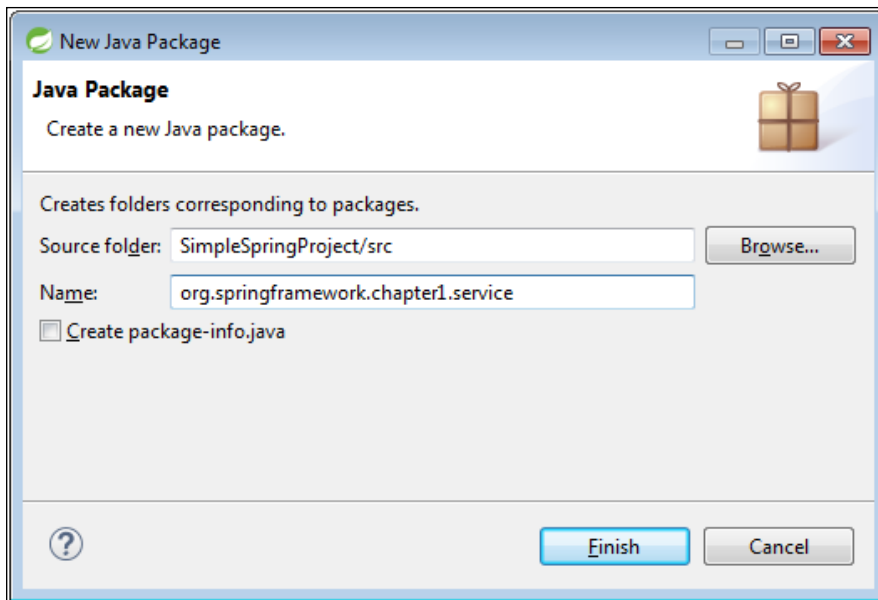
Now, you will have the content in your Project Explorer, as shown in the following screenshot:



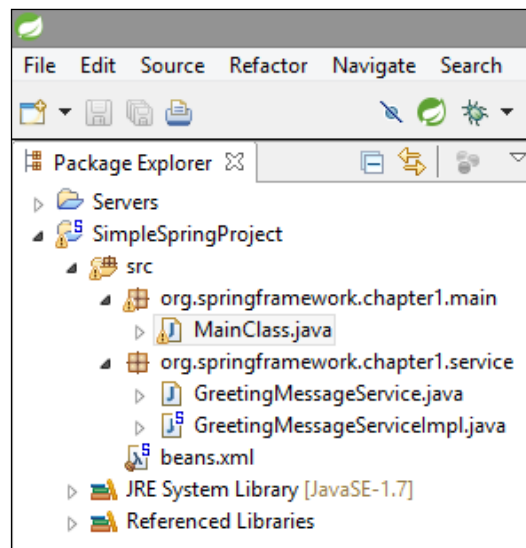
Creating source files

Now let's create the actual source files under the SimpleSpringProject project:

1. First, create the packages named `org.springframework.chapter1.service` and `org.springframework.chapter1.main`, as shown in the following screenshot. To do this, right-click on `src` in package explorer section and navigate to **New | Package**.



2. Create a class called `MainClass.java` inside the `org.springframework.chapter1.main` package. Then, create an interface named `GreetingMessageService.java` and its implementation class `GreetingMessageServiceImpl.java` inside the package `org.springframework.chapter1.service`, as shown in the following screenshot:



The following is the content of interface `GreetingMessageService.java` and its implementation `GreetingMessageServiceImpl.java`:

- `GreetingMessageService.java`:

```
package org.springframework.chapter1.service;

public interface GreetingMessageService {
    public String greetUser();
}
```
- `GreetingMessageServiceImpl.java`:

```
package org.springframework.chapter1.service;

import org.springframework.stereotype.Service;


@Service
public class GreetingMessageServiceImpl implements
GreetingMessageService {

    public String greetUser() {
        return "Welcome to Chapter-1 of book Learning
Spring Application Development";
    }

}
```

The `GreetingMessageService` interface has a `greetUser()` method. The `GreetingMessageServiceImpl` class implements the `GreetingMessageService` interface and provides definition to the `greetuser()` method. This class is annotated with the `@Service` annotation, which will define this class as service class

[



Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you

]

The following is the content of the file `MainClass.java`:

```
package org.springframework.chapter1.main;

import org.springframework.chapter1.service.
GreetingMessageService;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class MainClass {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext(
            "beans.xml");
        GreetingMessageService greetingMessageService =
context.getBean(
            "greetingMessageServiceImpl",
GreetingMessageService.class);
        System.out.println(greetingMessageService.greetuser());
    }
}
```

In `MainClass.java`, we are creating `ApplicationContext` using framework API, as shown in the following:

```
ApplicationContext context = new ClassPathXmlApplicationContext(
    "beans.xml");
```

This API loads Spring beans configuration file name `beans.xml`, which takes care of creating and initializing all the bean objects. We use the `getBean()` method of the created `ApplicationContext` to retrieve required Spring bean from the application context, as shown in the following:

```
GreetingMessageService greetingMessageService = context.getBean(
    "greetingMessageServiceImpl", GreetingMessageService.class);
```

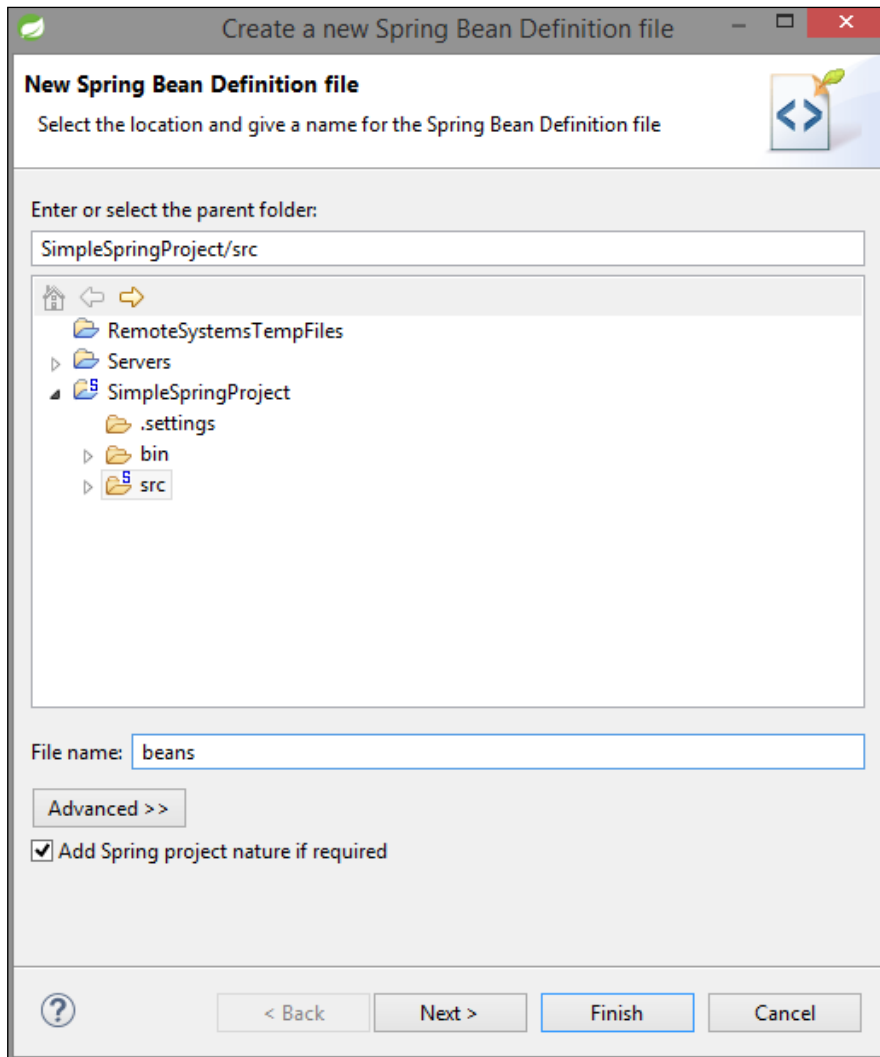
The `getBean()` method uses bean ID and bean class to return a bean object.

Creating the Spring bean configuration file

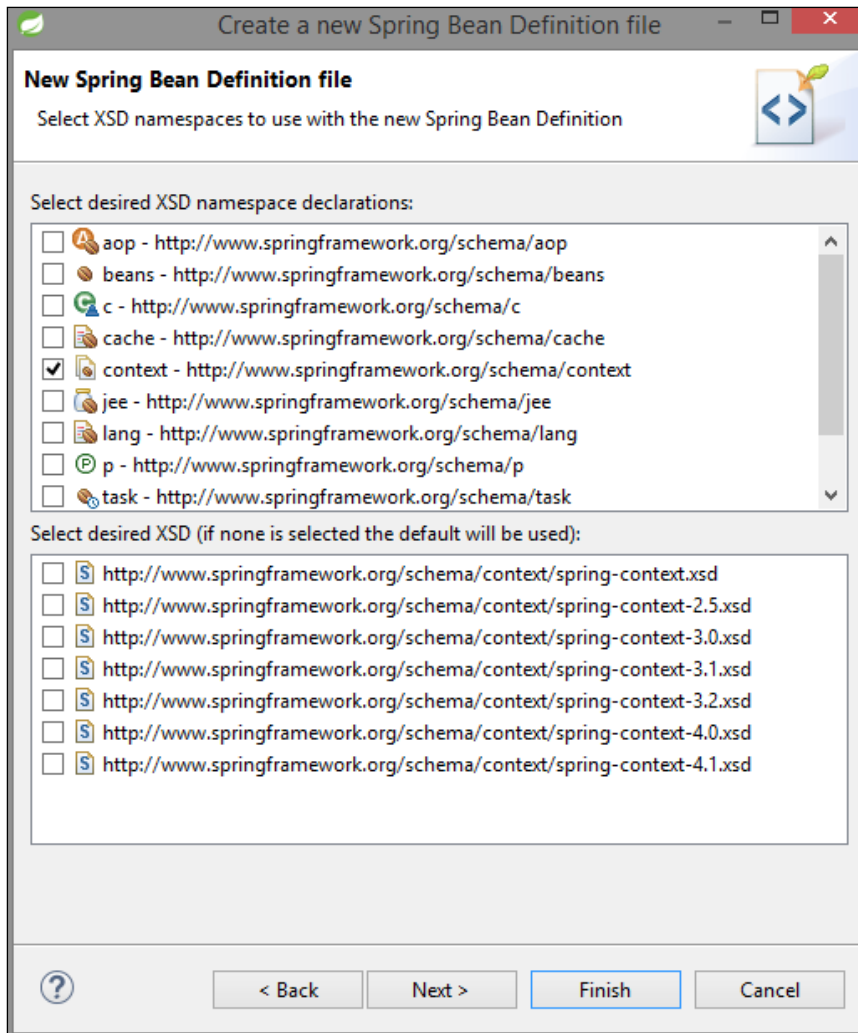
The Spring bean configuration file is used to configure the Spring beans in the Spring IoC container. As we have annotated the `GreetingMessageServiceImpl` class with `@Service` annotation, the next step is to add `<context:component-scan>` in the bean configuration file. To do this, follow these step

1. Create a Spring Bean Configuration file under the `src` directory. To do this, right-click on `src` in package explorer section and then navigate to **New | Spring Bean Configuration File**.

2. Enter the bean name `beans` and click on **Next**, as shown in the following screenshot:



3. Select the context option and click on **Finish**, as shown in the following screenshot:



4. Now the Spring bean configuration file is created. Add the following code to create an entry. The contents of the `beans.xml` file are as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

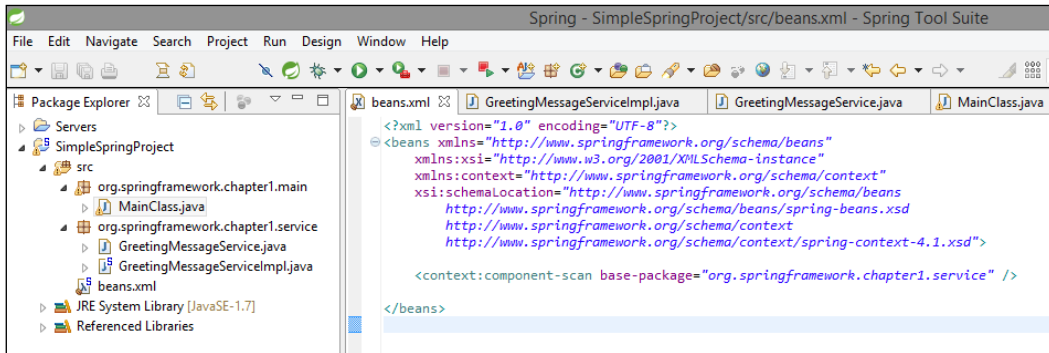


```
xmlns:context="http://www.springframework.org/schema/
context"
xsi:schemaLocation="http://www.springframework.org/
schema/beans
http://www.springframework.org/schema/beans/spring-
beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context-4.1.xsd">

<context:component-scan base-
package="org.springframework.chapter1.service"/>

</beans>
```

When the Spring application gets loaded into the memory, in order to create all the beans, the framework uses the preceding configuration file, as shown in the following screenshot:



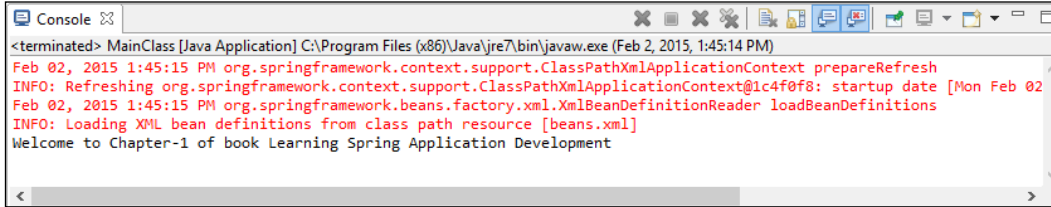
The Spring bean configuration file can be named anything, but developers usually keep the name `beans.xml`. This Spring bean configuration file should be available in classpath.



Running the program

Once you are done with creating source files and beans configuration files, you are ready for the next step, that is, compiling and running your program.

To execute the example, run the `MainClass.java` file. Right-click on `MainClass.java` and navigate to **Run As | Java Application**. If everything goes fine, then it will print the following message in STS IDE's console, as shown in the following screenshot:



```
<terminated> MainClass [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Feb 2, 2015, 1:45:14 PM)
Feb 02, 2015 1:45:15 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1c4f0f8: startup date [Mon Feb 02
Feb 02, 2015 1:45:15 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beans.xml]
Welcome to Chapter-1 of book Learning Spring Application Development
```

We have successfully created our first Spring application, where you learned how to create the Spring project and executed it successfully. We will see detailed examples in the next chapter.

Exercise

- Q1. What is Spring?
- Q2. List some of the features of Spring.
- Q3. Explain different modules in the Spring Framework.



The answers to these are provided in *Appendix A, Solution to Exercises*.

Summary

In this chapter, you were introduced the Spring Framework and acquainted with its features. You took a look at the versions of Spring. Then, you studied the architecture, and different modules in the Spring Framework such as the Spring Core Container, Spring AOP, Spring data access/integration, and the Spring Web module and Test module. You also understood the benefits of the Spring Framework. Finally, you created an application in Spring and took a look on package structure of Spring.

In the next chapter, we'll explore IoC, Dependency Injection, and Spring Core Container service. We'll also see bean's life cycle and bean's scope.

2

Inversion of Control in Spring

In this chapter, we'll explore the concept of **Inversion of Control (IoC)**. We'll then explore Spring Core Container, `BeanFactory`, and `ApplicationContext`, and you will learn how to implement them. We will take a look at **Dependency Injection (DI)** in Spring and their types: setter and constructor. We will wire beans using setter- and constructor-based Dependency Injection for different data types. We will also go through bean definition inheritance in Spring. We will then see autowiring in Spring and their modes. We will also see Spring bean's scope and its implementation. Then, we will move on to the life cycle of Spring bean.

The following is a list of topics that will be covered in this chapter:

- Understanding IoC
- Spring Container
- `BeanFactory`
- `ApplicationContext`
- Dependency Injection
- Constructor-based Dependency Injection
- Setter-based Dependency Injection
- Bean definition inheritance
- Autowiring in Spring
- Bean's scope
- Singleton
- Prototype
- Request
- Session
- Global-session

- Spring bean life cycle
- Initialization callback
- Destruction callback

Let's understand Inversion of Control.

Understanding Inversion of Control

In software engineering, IoC is a programming technique in which object coupling is bound at runtime by an assembler object and is usually not known at compile time using static analysis.

IoC is a more general concept, whereas DI is a concrete design pattern.

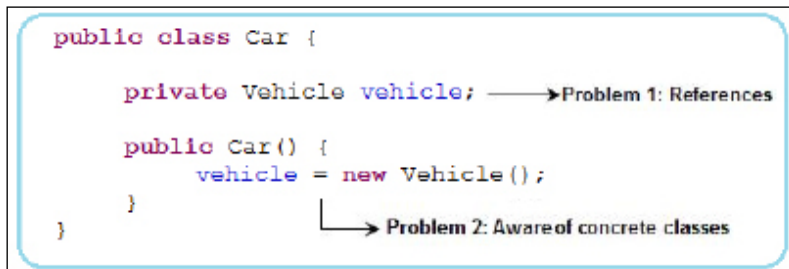
IoC is a way of thinking; a mechanism is required to activate components that provide specific functionality, due to which IoC depends on DI. The IoC pattern inverts responsibility of the managing the life cycle from the application to the framework, which makes writing Java applications even easier. IoC makes your code more manageable, more testable, and more portable. IoC also keeps component dependencies, life cycle events, and configuration outside of the components

Consider the following example: we have a `Car` class and a `Vehicle` class object. The biggest issue with the code is tight coupling between classes. In other words, the `Car` class depends on the `Vehicle` object. So, for any reason, changes in the `Vehicle` class will lead to the changes in, and compilation of, the `Car` class too.

So let's put down the problems with this approach:

- The biggest problem is that the `Car` class controls the creation of the `Vehicle` object
- The `Vehicle` class is directly referenced in the `Car` class, which leads to tight coupling between the `car` and `Vehicle` objects

The following figure illustrates this

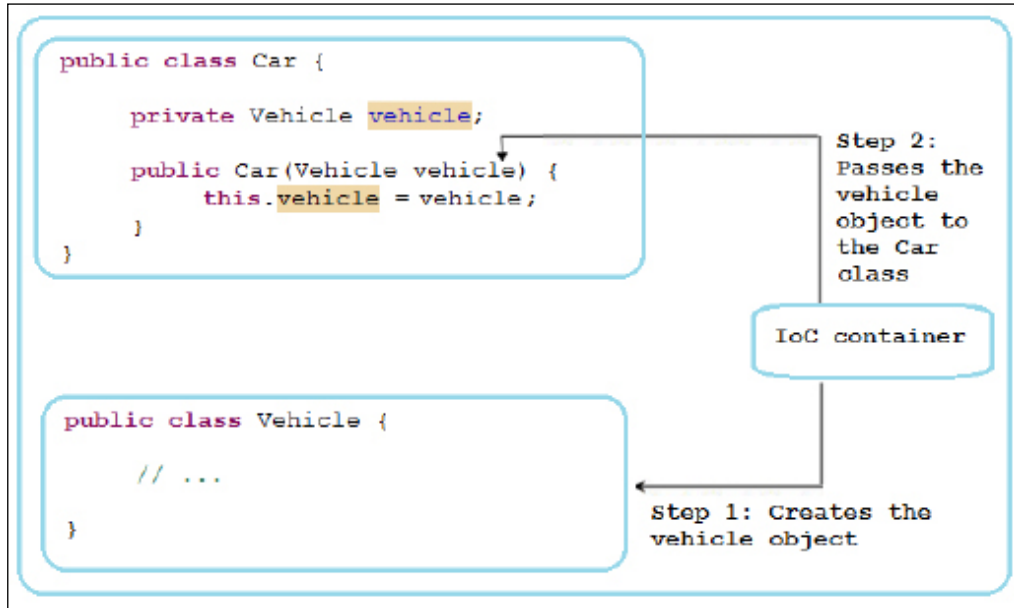


```
public class Car {  
    private Vehicle vehicle; —————> Problem 1: References  
    public Car() {  
        vehicle = new Vehicle();  
    }  
} —————> Problem 2: Aware of concrete classes
```

If, for any reason, the `vehicle` object is not created, the whole `Car` class will fail in the constructor initialization stage. The basic principle of IoC stands on the base of the Hollywood principle: Do not call us; we'll call you.

In other words, it's like the `Vehicle` class saying to the `Car` class, "don't create me, I'll create myself using someone else".

The IoC framework can be a class, client, or some kind of IoC container. The IoC container creates the `vehicle` object and passes this reference to the `Car` class, as shown here:



What is a container

In software development terminology, the word "container" is used to describe any component that can contain other components inside it. For example, Tomcat is a web container to contain deployed WAR files. JBoss is an application server container; it contains an EJB container, web container, and so on.

The container first creates the objects and then wires them together, after which it moves on to configure them, and finally manage their complete life cycle. It identifies the object dependencies, creates them, and then injects them into the appropriate objects.

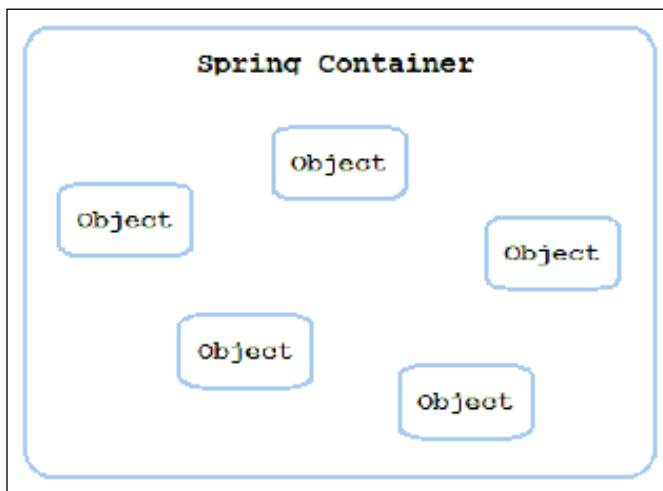
So, we can think about a container as an intermediate who'll register `vehicle` and `car` objects as separate entities, create the `vehicle` and `car` objects, and inject the `vehicle` object into `car`.

Spring Container

Spring Container is the central component of the Spring Framework. Spring Container manages the life cycle of an application's bean, which will live within Spring Container. Spring Container is responsible for wiring an application's beans by associating different beans together. Spring Container manages the components of applications using DI. The configuration metadata, which can be represented in XML, Java annotations, or Java code, helps Spring Container to decide the object to initiate, configure, and assemble

Let's take an example of Tomcat, which is a Servlet container. Tomcat creates the Servlet objects, which are required in order to run an application. While deploying an application, we configure all Servlets in an XML file. Tomcat reads this XML file identifies the Servlet to be instantiated, and then creates the identified Servlet

Spring is a container but not a container of Servlet. It is a container of beans and behaves as a factory of beans. So, we can have Spring Container and we can have as many objects as we want, as shown in the following diagram. Also, all these objects are managed by Spring Container. The container handles the instantiation of object, their whole life cycle, and finally their destruction too



Beans

Beans are reusable software components that are managed by the Spring IoC container. It contains the properties, setter, and getter methods of a class.

The Spring IoC container is represented by the interface `org.springframework.context.ApplicationContext`, which is responsible for instantiating, configuring, and assembling beans. Beans are reflected in the configuration metadata used by a container. The configuration metadata defines the instruction for the container and the objects to instantiate, configure, and assemble. This configuration metadata can be represented in XML, Java annotations, or Java code. In this chapter, we will configure using XML, which has been the traditional format to define configuration metadata. Refer to *Chapter 9, Inversion of Control in Spring – Using Annotation*, which is available online, on instructing the container to use Java annotations by providing a small amount of the XML configuration

XML-based bean configuration

The bean configuration information is stored in an XML file, which is used to create a bean definition using the `<bean>...</bean>` element. The bean definition contains the following metadata, which represents the configuration information of a bean

- A fully qualified class name that represents bean name
- The behavioral configuration elements, such as scope, life cycle, and so on, describe the bean's behavior in the Spring IoC container.

The following code snippet shows the basic structure of the XML configuration of the metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="..." class="...">
    <!-- configuration for this bean here -->
  </bean>
  <!-- more bean definitions here -->

</beans>
```


The configuration files have `<beans>` as the root element. The beans element has all other individual beans configured using the `<bean>` tag. Every `<bean>` tag needs to specify a class attribute and can have an optional ID or name attribute. The ID attributes enforce uniqueness in naming the beans. The class attribute has the fully classified class name; for example, the `src.org.packt.Spring.chapter2.Employee` class can be configured as follows

```
...
<bean id="employeeBean"
      class="src.org.packt.Spring.chapter2.Employee">
  </bean>
...
```

A reference of the `Employee` class instance is returned when the configuration file loaded using the `BeanFactory` or `ApplicationContext` container, and `employeeBean` is accessed using the `getBean (employeeBean)` method. The Spring IoC container is responsible for instantiating, configuring, and retrieving your Spring beans. The Spring IoC container enforces DI in its various forms and employs a number of established design patterns to achieve this.

Spring provides the following two interfaces that act as containers:

- `BeanFactory`: This is a basic container, and all other containers implement `BeanFactory`.
- `ApplicationContext`: This refers to the subinterface of `BeanFactory` and is mostly used as a container in enterprise applications.

To instantiate Spring Container, create an object of any of the `BeanFactory` or `ApplicationContext` implementation classes that supply the Spring bean configuration. The basic packages in the Spring IoC container of the Spring Framework are `org.springframework.beans` and `org.springframework.context`. An advanced configuration mechanism is provided by the `BeanFactory` interface to manage any type of object. The `ApplicationContext` interface implements the `BeanFactory` interface, which provides enterprise-specific functionality and supports message-resource handling, Spring's AOP features, event publication, and `WebApplicationContext` for use in web applications.

Both the containers, `BeanFactory` and `ApplicationContext`, are responsible for providing DI. For all the configured beans, these containers act as a repository. These containers initiate a registered bean, populate the bean's properties, and call the `init()` method to make the bean ready for use. The `destroy()` method of bean is invoked during the shutdown of the application. The `init()` and `destroy()` methods reflect the Servlet life cycle, where initialization can be performed during the `init()` method and cleanup during the `destroy()` method.

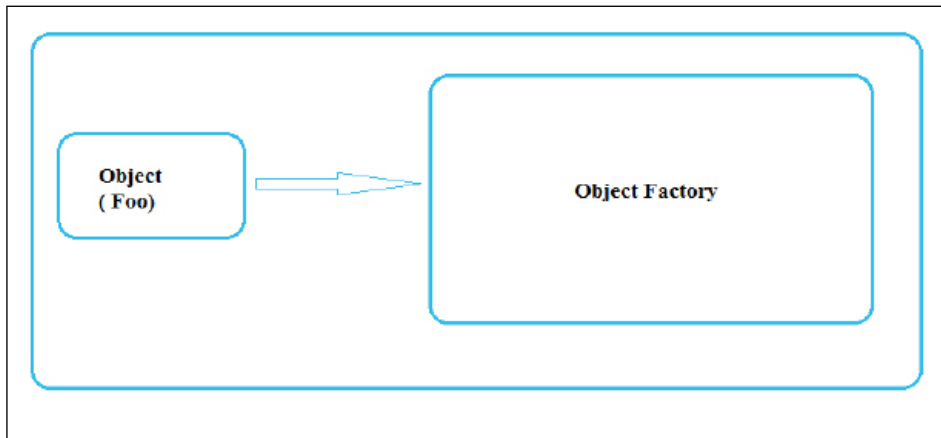
BeanFactory

Spring creates all the instances, along with the references to the objects you require. This is different from when you create an instance yourself with the help of the `new` method. This is called a factory pattern.

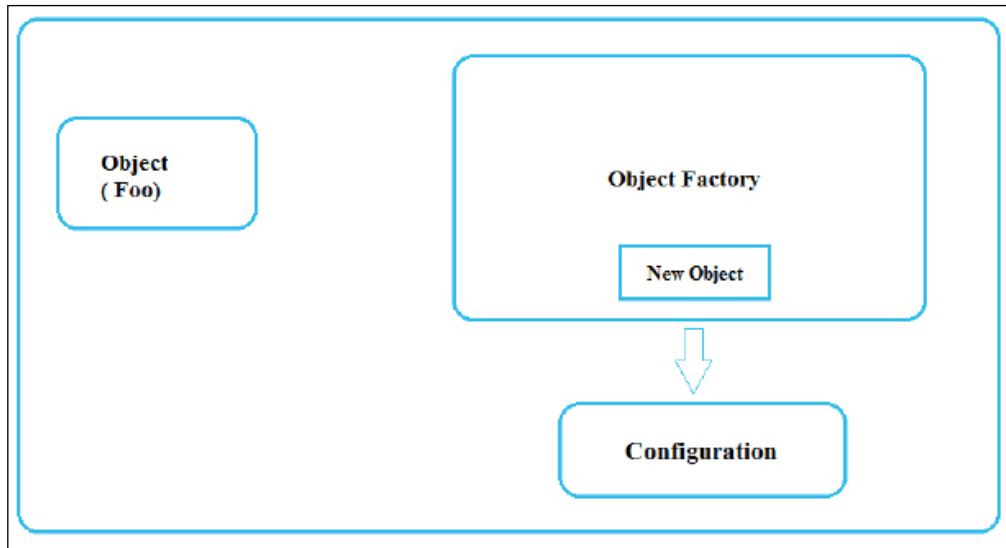
What is a factory pattern?

In a factory pattern, we have an object that behaves as the object factory. Basically, if you need an instance of any object, you don't have to create the instance yourself. Instead, you call a method of this factory, which then returns the instance you wanted. This factory reads from a configuration file, which acts as a blueprint that contains guidelines on how we can create the object.

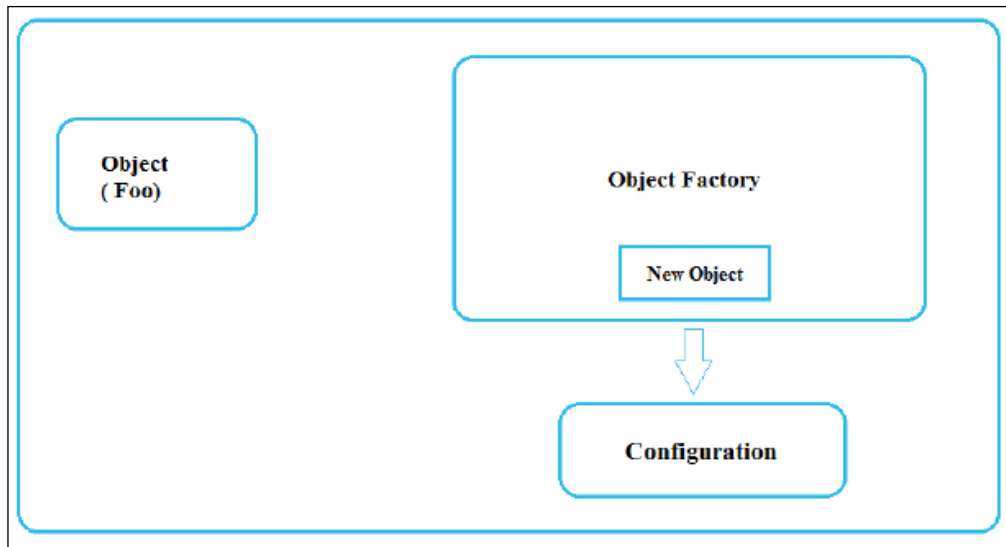
Assume that we have an object `Foo` and instead of creating a new object `Bar`, we make a call to another Java object, which is a `Factory` object. The job of the `Factory` object is to create and hand over a new object `Bar` to the object `Foo`, as shown in the following figure. The whole purpose of this factory is to produce objects



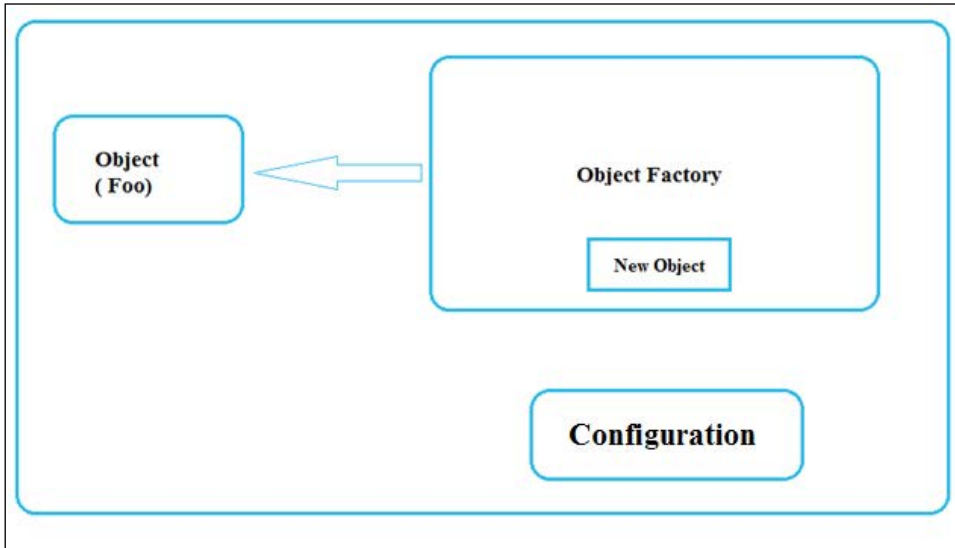
The `Factory` object reads from the configuration, which has metadata with details about the object that needs to be created. Configuration is a blueprint of all those objects that `Factory` creates. The `Factory` object reads from this configuration file as shown here:



The `Foo` object interacts with the `Factory` object to get an object with a certain specification. The `Factory` object finds out what the blueprint for that particular object specification is and then creates a new object, as shown in the following figure



Once the object has been created, `Factory` hands back the requesting `Bar` object to the `Foo` object. So, now `Foo` will have a new object it wants not using `new()` but using `Factory`, as shown in the following figure. This is something that Spring does.

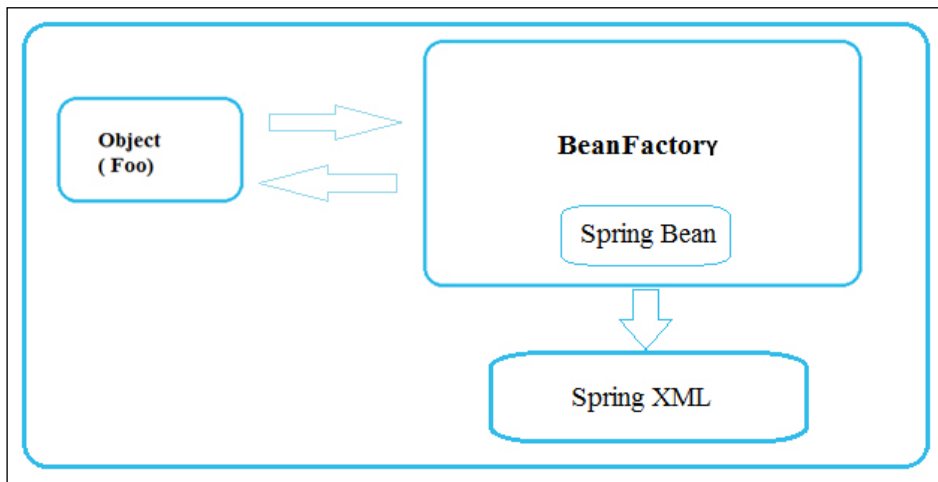


Spring BeanFactory

Spring has objects of the `BeanFactory` type that behave like the `Factory` object. You specify the blueprints object in a configuration file, which is an XML file, and then supply it to `BeanFactory`. Later, if you need the instance of any object, you can ask `BeanFactory` for it, which then refers to the XML file and constructs the bean as specified. This bean is now a Spring bean as it has been created by Spring Container and is returned to you. Let's now summarize this:

1. Spring has `BeanFactory`, which creates new objects for us. So, the `Foo` object will call `BeanFactory`.
2. `BeanFactory` would read from Spring XML, which contains all the bean definitions. Bean definitions are the blueprints here `BeanFactory` will create beans from this blueprint and then make a new Spring bean.

3. Finally, this new Spring bean is handed back to `Foo`, as shown here:



The advantage here is that this new bean has been created in this `BeanFactory`, which is known by Spring. Spring handles the creation and the entire life cycle of this bean. So, in this case, Spring acts as container for this newly created Spring bean.

`BeanFactory` is defined by the `org.springframework.beans.factory.BeanFactory` interface. The `BeanFactory` interface is the central IoC container interface in Spring and provides the basic end point for Spring Core Container towards the application to access the core container service.

It is responsible for containing and managing the beans. It is a factory class that contains a collection of beans. It holds multiple bean definitions within itself and then instantiates that bean as per the client's demands.

`BeanFactory` creates associations between collaborating objects as they're instantiated. This removes the burden of configuration from the bean itself along with the bean's client. It also takes part in the life cycle of a bean and makes calls to custom initialization and destruction methods.

Implementation of BeanFactory

There are many implementations of the `BeanFactory` interface, with the `org.springframework.beans.factory.xml.XmlBeanFactory` class being the most popularly used one, which reads the bean definition and initiates them based on the definitions contained in the XML file. Depending on the bean definition, the factory will return either an independent instance or a single shared instance of a contained object.

This class has been deprecated in favor of `DefaultListableBeanFactory` and `XmlBeanDefinitionReader`, and the purpose of this implementation is just to explain `BeanFactory`. The constructor for `XmlBeanFactory` takes an implementation of the `Resource` interface as an argument, as shown in the following line of code:

```
XmlBeanFactory (Resource resource)
```

The `Resource` interface has many implementations. The two commonly used implementations are shown in the following table:

The Resource interfaces	Description
<code>org.springframework.core.io.FileSystemResource</code>	This loads the configuration file from the underlying filesystem
<code>org.springframework.core.io.ClassPathResource</code>	This loads the configuration file from the classpath

Let's assume that beans are configured in the `beans.xml` file located in the C drive

```
...
<bean id="mybean" class="...">
    ...
</bean>
...
```

The code snippet to load the configuration file using `BeanFactory` is given as follows:

```
BeanFactory bfObj = new XmlBeanFactory (new FileSystemResource
("c:/beans.xml"));

MyBean beanObj= (MyBean) bfObj.getBean ("mybean");
```

Here, we've used `FileSystemResource`, which is one of the `Resource` interface implementations. The `bfObj` object corresponds to Spring Container, one that has loaded the bean definitions from the `beans.xml` file. `BeanFactory` is a lazy container, so at this point, only bean definitions get loaded, but beans themselves are not instantiated yet. At the second line, we call the `getBean()` method of the `BeanFactory` object created by passing the bean ID "mybean" as an argument to this method.

`BeanFactory` reads the bean definition of a bean with the ID "mybean" from Spring's `beans.xml` file, instantiates it, and then returns a reference.

The `BeanFactory` interface has different methods, such as `getBean`, `containsBean`, and so on, for client code to call. You can get the complete list of these methods from <http://docs.spring.io/spring/docs/2.0.x/reference/beans.html>.

The `BeanFactory` container is usually used in very simple applications; however, in real-time projects, the `ApplicationContext` container is used.

ApplicationContext

Like `BeanFactory`, `ApplicationContext` is also used to represent Spring Container, built upon the `BeanFactory` interface. `ApplicationContext` is suitable for Java EE applications, and it is always preferred over `BeanFactory`. All functionality of `BeanFactory` is included in `ApplicationContext`.

The `org.springframework.context.ApplicationContext` interface defines `ApplicationContext`. `ApplicationContext` and provides advanced features to our Spring applications that make them enterprise-level applications, whereas `BeanFactory` provides a few basic functionalities. Let's discuss them:

- Apart from providing a means of resolving text messages, `ApplicationContext` also includes support for i18n of those messages.
- A generic way to load file resources, such as images, is provided by `ApplicationContext`.
- The events to beans that are registered as listeners can also be published by `ApplicationContext`.
- `ApplicationContext` handles certain operations on the container or beans in the container declaratively, which have to be handled with `BeanFactory` in a programmatic way.
- It provides `ResourceLoader` support. This is used to handle low-level resources, Spring's `Resource` interface, and a flexible generic abstraction. `ApplicationContext` itself is `ResourceLoader`. Hence, access to deployment-specific `Resource` instances is provided to an application.
- It provides `MessageSource` support. `MessageSource`, an interface used to obtain localized messages with the actual implementation being pluggable, is implemented by `ApplicationContext`.

Implementation of ApplicationContext

The most commonly used `ApplicationContext` implementations are as follows:

- `ClassPathXmlApplicationContext`: This bean definition is loaded by the container from the XML file that is present in the classpath by treating context definition files as classpath resources. `ApplicationContext` can be loaded from within the application's classpath using `ClassPathXmlApplicationContext`:

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("spring-beans.xml");
```

- `FileSystemXmlApplicationContext`: This bean definition is loaded by the container from an XML file. Here, the full path of the XML bean configuration file should be provided to the constructor:

```
ApplicationContext context =
    new FileSystemXmlApplicationContext("classpath:beans.xml");
```

In the preceding code snippet, the `ApplicationContext` instance is created using the `FileSystemXmlApplicationContext` class and `beans.xml` is specified as a parameter.

The `getBean()` method can be used to access a particular bean by specifying its ID, as shown in following code snippet:

```
MyBean beanObj= (MyBean) context.getBean("mybean");
```

In the preceding code snippet, the `getBean()` method accepts the ID of the bean and returns the object of the bean.

`ApplicationContext` is an active container that initiates all the configured beans as soon as the `ApplicationContext` instance is created and before the user calls the `getBean()` method. The advantage of this active creation of beans by `ApplicationContext` is the handling of exceptions during the startup of the application itself.

- `XmlWebApplicationContext`: This is used to create the context in web application by loading configuration the XML file with definitions of all beans from standard locations within a web application directory. The default location of the configuration XML file is `/WEB-INF/applicationContext.xml`.
- `AnnotationConfigApplicationContext`: This is used to create the context by loading Java classes annotated with the `@Configuration` annotation instead of XML files. The `AnnotationConfigApplicationContext` class is used when we define Java-based Spring bean configuration for the bean definition instead of XML files.

- `AnnotationConfigWebApplicationContext`: This is used to create the web application context by loading the Java classes annotated with the `@Configuration` annotation instead of XML files in the web application.

To demonstrate an implementation of `ApplicationContext`, an example of `PayrollSystem` can be considered. It will have the `EmployeeService` interface, `EmployeeServiceImpl` class, and `PayrollSystem` class with the main method.

In the `EmployeeService.java` interface, you'll find the following code

```
package org.packt.Spring.chapter2.ApplicationContext;  
  
public interface EmployeeService {  
  
    public Long generateEmployeeId();  
  
}
```

The `EmployeeService.java` interface is a plain old Java interface that has a method named `generateEmployeeId()` to generate a unique employee ID on each call.

In the `EmployeeServiceImpl.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.ApplicationContext;  
  
public class EmployeeServiceImpl implements EmployeeService {  
  
    @Override  
    public Long generateEmployeeId() {  
        return System.currentTimeMillis();  
    }  
  
}
```

The `EmployeeServiceImpl.java` class implements the `EmployeeService` interface. This `generateEmployeeId()` class-implemented method is used to generate a unique employee ID on each part of this method based on the system's current time.

In the `PayrollSystem.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.ApplicationContext;  
  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplication  
Context;  
  
public class PayrollSystem {
```

```
public static void main(String[] args) {

    ApplicationContext context = new
    ClassPathXmlApplicationContext(
        "beans.xml");

    EmployeeService empService = (EmployeeServiceImpl)
context
        .getBean("empServiceBean");
    System.out.println("Unique Employee Id: " +
empService.generateEmployeeId());
}

}
```

The `PayrollSystem.java` class is a main class that contains the `main()` method. This method creates an instance of `ApplicationContext`, calls the `getBean()` method to get the bean of `EmployeeService`, and then prints the generated unique employee ID by calling the method from this bean.

The `beans.xml` file contains the bean definition for `EmployeeServiceImpl`, as shown in the following code snippet:

```
...
<bean id="empServiceBean"
class="org.packt.Spring.chapter2.ApplicationContext.
EmployeeServiceImp">
</bean>
...
```

When you successfully run `PayrollSystem.java`, the output will be printed on the console as follows:

```
Unique Employee Id: 1401215855074
```

The generated `Employee Id` value will be different for you when you run the preceding code in your local system as it is based on the current time.

A Spring application requires several beans or objects to work together in order to develop a loosely coupled application. Objects depend on each other to carry out their respective functions and this applies to beans too. Now let's understand DI.

Dependency Injection

Dependency Injection (DI) is a design pattern in which an object's dependency is injected by the framework rather than by the object itself. It reduces coupling between multiple objects as it is dynamically injected by the framework. In DI, the framework is completely responsible for reading configuration

The advantages of DI are as follows:

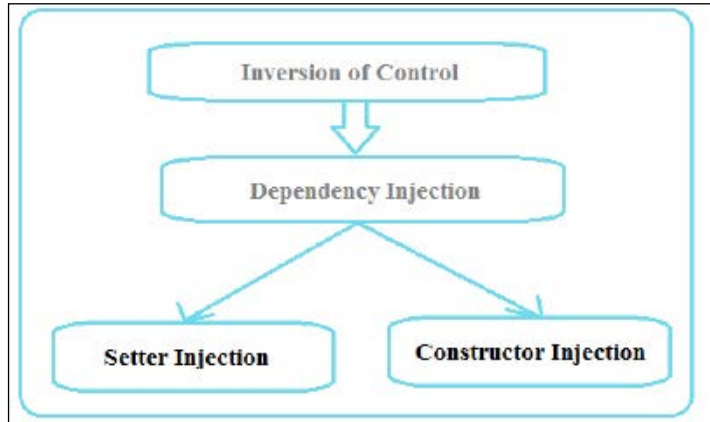
- Loosely coupled architecture.
- Separation of responsibility.
- Configuration and code are separate.
- A different implementation can be supplied using configuration without changing the code dependent.
- Improves testability.
- DI allows you to replace actual objects with mock objects. This improves testability by writing simple JUnit tests that use mock objects.

Dependency Injection in Spring

In the Spring Framework, DI is used to satisfy the dependencies between objects. It exists in only two types:

- **Constructor Injection:** By invoking a constructor containing a number of arguments, constructor-based DI can be accomplished. These arguments are injected at the time of instance instantiation.
- **Setter Injection:** Setter-based DI is attained by calling setter methods on your beans. Using setter methods defined in a Spring configuration file, the dependencies are "set" in the objects.

The following figure gives us a better picture



Let's consider an example where the `EmployeeServiceImpl` class has an instance field `employeeDao` of the `EmployeeDao` type, a constructor with an argument, and a `setEmployeeDao` method.

In the `EmployeeServiceImpl.java` class, you'll find the following code

```
public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDao employeeDao;

    public EmployeeServiceImpl(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }

    public void setEmployeeDao(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }
}
```

In the `EmployeeDaoImpl.java` class, you'll find the following code

```
public class EmployeeDaoImpl implements EmployeeDao {

    // ...

}
```

Here, an instance of `EmployeeDao` can be provided by the configuration file by either the constructor method or the setter method. Before we understand what these are in more detail, let's understand how generally two objects interact with each other to make an even more meaningful object.

The Has-A relationship

When a class contains another class as instance field; for example, the `EmployeeServiceImpl` class contains `EmployeeDao` as its field. This is called a Has-A relationship since we say, "EmployeeServiceImpl has an EmployeeDao". So, without `employeeDao`, `EmployeeServiceImpl` cannot perform. The following code illustrates this:

```
public class EmployeeServiceImpl implements EmployeeService {  
  
    private EmployeeDao employeeDao = null;  
  
}
```

So, `employeeDao` is the dependency that needs to be resolved in order to make `EmployeeServiceImpl` fully functional. The way to create an object of the `EmployeeDao` type or, in other words, satisfy the dependency of `EmployeeServiceImpl` in Java is shown here:

```
public class EmployeeServiceImpl implements EmployeeService {  
  
    private EmployeeDao employeeDao = null;  
  
    public EmployeeServiceImpl() {  
        this.employeeDao = new EmployeeDaoImpl();  
    }  
  
    public void setEmployeeDao() {  
        this.employeeDao = new EmployeeDaoImpl();  
    }  
}
```

It is not a very good option as once the `EmployeeServiceImpl` object is created, you don't have any way to have the object of the `employeeDao` type swapped with a subclass implementation.

Constructor-based Dependency Injection

Constructor Injection is the process of injecting the dependencies of an object through its constructor argument at the time of instantiating it. In other words, we can say that dependencies are supplied as an object through the object's own constructor. The bean definition can use a constructor with zero or more arguments to initiate the bean, as shown here:

```
public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDao employeeDao = null;

    public EmployeeServiceImpl(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }
}
```

In the preceding code, the object of the `EmployeeDao employeeDao` type is injected as a constructor argument to the `EmployeeServiceImpl` class. We need to configure bean definition in the configuration file that will perform Constructor Injecti

The Spring bean XML configuration tag `<constructor-arg>` is used for Constructor Injection:

```
...
<bean id="employeeService"
    class="org.packt.Spring.chapter2.dependencyinjection.
EmployeeServiceImpl">
    <constructor-arg ref="employeeDao" />
</bean>

<bean id="employeeDao"
    class="org.packt.Spring.chapter2.dependencyinjection.
EmployeeDaoImpl">
    </bean>
...
```

In the preceding code snippet, there is a Has-A relationship between the classes, which is `EmployeeServiceImpl HAS-A EmployeeDao`. Here, we inject a user-defined objec as the source bean into a target bean using Constructor Injection. Once we have the `employeeDao` bean to inject it into the target `employeeService` bean, we need another attribute called `ref` – its value is the name of the ID attribute of the source bean, which in our case is `"employeeDao"`.

The <constructor-arg> element

The <constructor-arg> subelement of the <bean> element is used for Constructor Injection. The <constructor-arg> element supports four attributes. They are explained in the following table:

Attributes	Description	Occurrence
index	It takes the exact index in the constructor argument list. It is used to avoid ambiguity such as when two arguments are of the same type.	Optional
type	It takes the type of this constructor argument.	Optional
value	It describes the content in a simple string representation, which is converted into the argument type using the PropertyEditors Java beans.	Optional
ref	It refers to another bean in this factory.	Optional

Constructor Injection – injecting simple Java types

Here, we inject simple Java types into a target bean using Constructor Injection.

The Employee class has employeeName as String, employeeAge as int, and married as boolean. The constructor initializes all these three fields

In the Employee.java class, you'll find the following code:

```
package org.packt.Spring.chapter2.constructioninjection.  
simplejavatype;  
  
public class Employee {  
  
    private String employeeName;  
    private int employeeAge;  
    private boolean married;  
  
    public Employee(String employeeName, int employeeAge, boolean  
married) {  
        this.employeeName = employeeName;  
        this.employeeAge = employeeAge;  
        this.married = married;  
  
    }  
    @Override  
    public String toString() {
```

```

        return "Employee Name: " + this.employeeName + " , Age:"
            + this.employeeAge + " , IsMarried: " +
married;
    }
}

```

In the `beans.xml` file, you'll find the following code:

```

...
<bean id="employee"
    class="org.packt.Spring.chapter2.constructioninjection
.simplejavatype.Employee">
    <constructor-arg value="Ravi Kant Soni" />
    <constructor-arg value="28" />
    <constructor-arg value="False" />

</bean>
...

```

Constructor Injection – resolving ambiguity

In the Spring Framework, whenever we create a Spring bean definition file and provide values to the constructor, Spring decides implicitly and assigns the bean's value in the constructor by means of following key factors:

- Matching the number of arguments
- Matching the argument's type
- Matching the argument's order

Whenever Spring tries to create the bean using Construction Injection by following the aforementioned rules, it tries to resolve the constructor to be chosen while creating Spring bean and hence results in the following situations.

No ambiguity

If no matching constructor is found when Spring tries to create a Spring bean using the preceding rule, it throws the `BeanCreationException` exception with the message: `Could not resolve matching constructor`.

Let's understand this scenario in more detail by taking the `Employee` class from earlier, which has three instance variables and a constructor to set the value of this instance variable.

The `Employee` class has a constructor in the order of `String`, `int`, and `boolean` to be passed while defining the bean in the definition file

In the `beans.xml` file, you'll find the following code

```
...
    <bean id="employee"
        class="org.packt.Spring.chapter2.constructioninjection
        .simplejavatype.Employee">
        <constructor-arg value="Ravi Kant Soni" />
        <constructor-arg value="False" />
        <constructor-arg value="28" />
    </bean>
...
```

If the orders in which `constructor-arg` is defined are not matching, then you will get the following error:

```
Exception in thread "main" org.springframework.beans.factory.
UnsatisfiedDependencyException:
Error creating bean with name employee defined in the classpath
resource [beans.xml]: Unsatisfied dependency expressed through
constructor argument with index 1 of type [int]: Could not convert
constructor argument value of type [java.lang.String] to required
type [int]: Failed to convert value of type 'java.lang.String' to
required type 'int'; nested exception is
java.lang.NumberFormatException: For input string: "False"
```

Solution – use index attribute

The solution to this problem is to fix the order. Either we modify the `constructor-arg` order of the bean definition file or we use the `index` attribute of `constructor-arg` as follows:

```
...
    <bean id="employee"
        class="org.packt.Spring.chapter2.constructioninjection
        .simplejavatype.Employee">
        <constructor-arg value="Ravi Kant Soni" index="0" />
        <constructor-arg value="False" index="2" />
        <constructor-arg value="28" index="1" />
    </bean>
...
```

Remember that the `index` attribute always starts with 0.

Parameter ambiguity

Sometimes, there is no problem in resolving the constructor, but the constructor chosen is leading to inconvertible data. In this case, `org.springframework.beans.factory.UnsatisfiedDependencyException` is thrown just before the data is converted to the actual type.

Let's understand this scenario in more depth; the `Employee` class contains two constructor methods and both accept three arguments with different data types.

The following code snippet is also present in `Employee.java`:

```
package
org.packt.Spring.chapter2.constructioninjection.simplejavatype;

public class Employee {

    private String employeeName;
    private int employeeAge;
    private String employeeId;

    Employee(String employeeName, int employeeAge, String
employeeId) {
        this.employeeName = employeeName;
        this.employeeAge = employeeAge;
        this.employeeId = employeeId;
    }

    Employee(String employeeName, String employeeId, int
employeeAge) {
        this.employeeName = employeeName;
        this.employeeId = employeeId;
        this.employeeAge = employeeAge;
    }

    @Override
    public String toString() {
        return "Employee Name: " + employeeName + ", Employee
Age: "
                + employeeAge + ", Employee Id: " +
employeeId;
    }
}
```

In the `beans.xml` file, you'll find the following code

```
...
    <bean id="employee"
        class="org.packt.Spring.chapter2.constructioninjection.
        simplejavatype.Employee">
        <constructor-arg value="Ravi Kant Soni" />
        <constructor-arg value="1065" />
        <constructor-arg value="28" />
    </bean>
...
```

Spring chooses the wrong constructor to create the bean. The preceding bean definition has been written in the hope that Spring will choose the second constructor as Ravi Kant Soni for `employeeName`, 1065 for `employeeId`, and 28 for `employeeAge`. But the actual output will be:

```
Employee Name: Ravi Kant Soni, Employee Age: 1065, Employee Id: 28
```

The preceding result is not what we expected; the first constructor is run instead of the second constructor. In Spring, the argument type 1065 is converted to `int`, so Spring converts it and takes the first constructor even though you assume it should be a string.

In addition, if Spring can't resolve which constructor to use, it will prompt the following error message:

```
constructor arguments specified but no matching constructor
found in bean 'CustomerBean' (hint: specify index and/or
type arguments for simple parameters to avoid type ambiguities)
```

Solution – use type attribute

The solution to this problem is to use the `type` attribute to specify the exact data type for the constructor:

```
...
    <bean id="employee"
        class="org.packt.Spring.chapter2.constructioninjection.
        simplejavatype.Employee">

        <constructor-arg value="Ravi Kant Soni"
type="java.lang.String"/>
        <constructor-arg value="1065" type="java.lang.String"/>
    </bean>
...
```

```
        <constructor-arg value="28" type="int"/>
    </bean>
    ...
```

Now the output will be as expected:

```
Employee Name: Ravi Kant Soni, Employee Age: 28, Employee Id: 1065
```

The setter-based Dependency Injection

The setter-based DI is the method of injecting the dependencies of an object using the setter method. In the setter injection, the Spring container uses `setXXX()` of the Spring bean class to assign a dependent variable to the bean property from the bean configuration file. The setter method is more convenient to inject more dependencies since a large number of constructor arguments makes it awkward.

In the `EmployeeServiceImpl.java` class, you'll find the following code

```
public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDao employeeDao;

    public void setEmployeeDao(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }
}
```

In the `EmployeeDaoImpl.java` class, you'll find the following code

```
public class EmployeeDaoImpl implements EmployeeDao {
    // ...
}
```

In the preceding code snippet, the `EmployeeServiceImpl` class defined the `setEmployeeDao()` method as the setter method where `EmployeeDao` is the property of this class. This method injects values of the `employeeDao` bean from the bean configuration file before making the `employeeService` bean available to the application.

The Spring bean XML configuration tag `<property>` is used to configure properties. The `ref` attribute of property elements is used to define the reference of another bean.

In the `beans.xml` file, you'll find the following code:

```
...
<bean id="employeeService"
      class="org.packt.Spring.chapter2.dependencyinjection.
EmployeeServiceImpl">
    <property name="employeeDao" ref="employeeDao" />
</bean>

<bean id="employeeDao"
      class="org.packt.Spring.chapter2.dependencyinjection.
EmployeeDaoImpl">
</bean>
...
```

The <property> element

The <property> element invokes the setter method. The bean definition can be describing the zero or more properties to inject before making the bean object available to the application. The <property> element corresponds to JavaBeans' setter methods, which are exposed by bean classes. The <property> element supports the following three attributes:

Attributes	Description	Occurrence
name	It takes the name of Java bean-based property	Optional
value	It describes the content in a simple string representation, which is converted into the argument type using JavaBeans' PropertyEditors	Optional
ref	It refers to a bean	Optional

Setter Injection – injecting a simple Java type

Here, we inject string-based values using the setter method. The `Employee` class contains the `employeeName` field with its setter method

In the `Employee.java` class, you'll find the following code:

```
package org.packt.Spring.chapter2.setterinjection;

public class Employee {
```

```
String employeeName;

public void setEmployeeName(String employeeName) {
    this.employeeName = employeeName;
}

@Override
public String toString() {
    return "Employee Name: " + employeeName;
}
}
```

In the `beans.xml` file, you'll find the following code

```
...
<bean id="employee" class="org.packt.Spring.chapter2.
setterinjection.Employee">
    <property name="employeeName" value="Ravi Kant Soni" />
</bean>
...
```

In the preceding code snippet, the bean configuration file set the property value

In the `PayrollSystem.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.setterinjection;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
ApplicationContext;

public class PayrollSystem {

    public static void main(String[] args) {
        ApplicationContext context = new
        ClassPathXmlApplicationContext(
            "beans.xml");
        Employee employee = (Employee)
        context.getBean("employee");
        System.out.println(employee);
    }
}
```

The output after running the PayrollSystem class will be as follows:

```
INFO: Refreshing org.springframework.context.support.
ClassPathXmlApplicationContext
@1ba94d: startup date [Sun Jan 25 10:11:36 IST 2015]; root of
context hierarchy
Jan 25, 2015 10:11:36 AM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[beans.xml]
Employee Name: Ravi Kant Soni
```

Setter Injection – injecting collections

In the Spring IoC container, beans can also access collections of objects. Spring allows you to inject a collection of objects in a bean using Java's collection framework. Setter Injection can be used to inject collection values in the Spring Framework. If we have a dependent object in the collection, we can inject this information using the `ref` element inside the list, set, or map. Let's discuss them in more detail:

- `<list>`: This element describes a `java.util.List` type. A list can contain multiple bean, ref, value, null, another list, set, and map elements. The necessary conversion is automatically performed by `BeanFactory`.
- `<set>`: This element describes a `java.util.Set` type. A set can contain multiple bean, ref, value, null, another set, list, and map elements.
- `<map>`: This element describes a `java.util.Map` type. A map can contain zero or more `<entry>` elements, which describes a key and value.

The `Employee` class is a class with an injecting collection.

In the `Employee.java` class, you'll find the following code:

```
package org.packt.Spring.chapter2.setterinjection;

import java.util.List;
import java.util.Map;
import java.util.Set;

public class Employee {

    private List<Object> lists;
    private Set<Object> sets;
    private Map<Object, Object> maps;
```

```
public void setLists(List<Object> lists) {
    this.lists = lists;
}

public void setSets(Set<Object> sets) {
    this.sets = sets;
}

public void setMaps(Map<Object, Object> maps) {
    this.maps = maps;
}
}
```

The bean configuration file is the one that injects each and every property of the Employee class.

In the `beans.xml` file, you'll find the following code:

```
...
<bean id="employee" class="org.packt.Spring.chapter2.setterinjection.
Employee">
    <property name="lists">
        <list>
            <value>Ravi Kant Soni</value>
            <value>Shashi Kant Soni</value>
            <value>Shree Kant Soni</value>
        </list>
    </property>
    <property name="sets">
        <set>
            <value>Namrata Soni</value>
            <value>Rishi Raj Soni</value>
        </set>
    </property>
    <property name="maps">
        <map>
            <entry key="Key 1" value="Sasaram"/>
            <entry key="Key 2" value="Bihar"/>
        </map>
    </property>
</bean>
...
```


In the preceding code snippet, we injected values of all three setter methods of the `Employee` class. The `List` and `Set` instances are injected with the `<list>` and `<set>` tags. For the `map` property of the `Employee` class, we injected a `Map` instance using the `<map>` tag. Each entry of the `<map>` tag is specified with the `<entry>` tag that contains a key-value pair of the `Map` instance.

Injecting inner beans

Similar to the concept of inner classes in Java, it is also possible to define a bean inside another bean; for example, in an **Automated Teller Machine (ATM)** system, we can have a printer bean as an inner bean of the `ATM` class.

The following are the characteristics of inner beans in Spring:

- A bean can optionally be declared as an inner bean when it doesn't need to be shared with other beans.
- An inner bean is defined within the context of its enclosing bean.
- Typically, the inner bean does not have an ID associated with it because the inner bean will not be shared outside of its enclosing bean. We can associate an ID; however, the value of this ID attribute is ignored by Spring.
- The inner class is independent of the inner bean. Any class can be defined as an inner bean; for instance, a `Printer` class is not an inner class, but a printer bean is defined as an inner bean.
- The scope of an inner bean is always a prototype.

The limitations of using inner beans are as follows:

- It cannot be reused or shared with other beans
- In practice, it affects the readability of the configuration file

An `ATM` class has a `Printer` class. We'll declare the printer bean as an inner bean (inside the enclosing `ATM` bean) since the `Printer` class is not referenced anywhere outside the `ATM` class. The `printBalance()` method of `ATM` delegates the call to the `printBalance()` method of the printer. The printer bean will be declared as an inner bean and will then be injected into the `ATM` bean using Setter Injection.

The `ATM` class delegates the call to print the balance to the `Printer` class.

The following code snippet can also be found in `ATM.java`:

```
package org.packt.Spring.chapter2.setterinjection;

public class ATM {
```

```
private Printer printer;

public Printer getPrinter() {
    return printer;
}

public void setPrinter(Printer printer) {
    this.printer = printer;
}

public void printBalance(String accountNumber) {
    getPrinter().printBalance(accountNumber);
}
}
```

In the preceding code snippet, the ATM class has a Printer class as property and the setter, `getPrinter()`, and `printBalance()` methods.

In the `Printer.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.setterinjection;

public class Printer {

    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public void printBalance(String accountNumber) {

        System.out.println(message + accountNumber);
    }
}
```

In the preceding code snippet, the Printer class has the `printBalance()` method. It has a message property, and a setter method sets the message value from the bean configuration file

In the `beans.xml` file, you'll find the following code

```
...
<bean id="atmBean" class="org.packt.Spring.chapter2.
setterinjection.ATM">
```

```
        <property name="printer">
            <bean
class="org.packt.Spring.chapter2.setterinjection.Printer">
                <property name="message"
                    value="The balance information is
printed by Printer for the account number"></property>
            </bean>
        </property>

    </bean>
...

```

Here, we declare `atmBean`. We declare the printer bean as an inner bean by declaring inside the enclosing `atmBean`. The `id` attribute cannot be used outside the context of `atmBean` and hence hasn't been provided to the printer bean.

Injecting null and empty string values in Spring

We come across two cases while injecting null and empty string values.

Case 1 – injecting an empty string

We can pass an empty string as a value, as shown in the following code, which is like `setEmployeeName("")` in the Java code:

```
...
<bean id="employee"
class="org.packt.Spring.chapter2.setterinjection.Employee">
    <property name="employeeName" value=""></property>
</bean>
...

```

Case 2 – injecting a null value

We can pass a null value, as shown in the following code, which is like `setEmployeeName(null)` in the Java code:

```
...
<bean id="employee"
class="org.packt.Spring.chapter2.setterinjection.Employee">

```

```

        <property name="employeeName">
            <null />
        </property>
    </bean>
    ...

```

Bean definition inheritance

Bean definition inheritance means that you have lot of bean definition in the bean configuration file and you have something that is common across lots of bean. There is a common setter value that has to be initialized across multiple beans and only then bean definition inheritance can be used

You can have one parent bean that contains all of these common definitions inside it, and then you can inherit all the common bean definitions across the other bean. This parent bean, which has all the common definitions, can be a bean in itself. This parent bean can be made into abstract bean definitions, so there are no beans created for it, and all it does is for the purpose of templating a bean definition

From a parent bean definition, a child bean definition inherits configuration data and can override or add values, as required. In an XML-based configuration file a child bean definition is indicated using a parent attribute that specifies the parent bean as the value of this attribute. Refer to the following table for clarity:

Beans	Description
ParentBean	<code><bean id="pBean" class="ParentBean"></code>
ChildBean	<code><bean id="cBean" class="ChildBean" parent="pBean"></code>

ParentBean and ChildBean are explained as follows:

- **ParentBean:** This is a parent bean that is used as a template to create other beans. It would be referred to in the XML file with `id="pBean"`.
- **ChildBean:** This is a child bean that inherits from the parent bean defined earlier. The `parent="pBean"` specifies that this bean is inheriting the properties of the ParentBean bean.

The child bean must accept the parent bean's property values. The child bean definition inherits constructor argument values and property values from the parent bean definition. The child bean definition overrides the initialization method setting and destroys method setting from the parent bean definition.

Spring bean definition inheritance is not related with the Java class inheritance. A parent bean is defined as a template and child beans can inherit the required configuration from this parent bean.

Now, the following example illustrates bean definition inheritance.

In the `Employee.java` class, you'll find the following code:

```
package org.packt.Spring.chapter2.beaninheritance;

public class Employee {

    private int employeeId;
    private String employeeName;
    private String country;

    public void setEmployeeId(int employeeId) {
        this.employeeId = employeeId;
    }

    public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    @Override
    public String toString() {
        return "Employee ID: " + employeeId + " Name: " +
employeeName
                + " Country: " + country;
    }

}
```

In the preceding code snippet, the `Employee` class contains properties named `employeeName`, `employeeId`, `country`, and their corresponding setter method. This class has also overridden the `toString()` method.

The Spring bean configuration file `beans.xml`, where we defined the `indianEmployee` bean as a parent bean with the `country` property and its value. Next, an `employeeBean` bean has been defined as the child bean of `indianEmployee` using the `parent="indianEmployee"` parent attribute. The child bean inherits `country` properties from the parent bean and introduces two more properties, `employeeId` and `employeeName`.

In the `beans.xml` file, you'll find the following code

```
...
    <bean id="indianEmployee"
class="org.packt.Spring.chapter2.beaninheritance.Employee">
        <property name="country"
value="India"></property>
    </bean>

    <bean id="employeeBean" parent="indianEmployee">
        <property name="employeeId" value="1065"></property>
        <property name="employeeName" value="Ravi Kant
Soni"></property>
    </bean>
...
```

In the `PayrollSystem.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.beaninheritance;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
ApplicationContext;

public class PayrollSystem {

    public static void main(String[] args) {

        ApplicationContext context = new
ClassPathXmlApplicationContext(
            "beans.xml");

        // using 'employeeBean'
        Employee employeeA = (Employee)
context.getBean("employeeBean");
        System.out.println(employeeA);

        // using 'indianEmployee'
```

```
        Employee employeeB = (Employee)
context.getBean("indianEmployee");
        System.out.println(employeeB);
    }
}
```

When we run the PayrollSystem class, the result will be as follows:

```
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext
@1c4f0f8: startup date [Sun Jan 25 14:31:50 IST 2015]; root of
context hierarchy
Jan 25, 2015 2:31:51 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[beans.xml]
Employee ID: 1065 Name: Ravi Kant Soni Country: India
Employee ID: 0 Name: null Country: India
```

Here, the indianEmployee bean is able to instantiate. In the indianEmployee bean, we have only set the value for the country property, so other fields get the null value. In the employeeBean, we have set only two properties, which are employeeId and employeeName, and the country property is inherited from the indianEmployee bean, so all the fields get their value for employeeBean.

Inheritance with abstract

Inheritance with abstract helps in creating a bean definition as a template, which cannot be instantiated and serves as a parent definition for child definitions. While defining a bean definition as a template, you should specify only the abstract attribute with the value true, for example, abstract="true".

In the beans.xml file, you'll find the following code

```
...
<bean id="indianEmployee"
class="org.packt.Spring.chapter2.beaninheritance.Employee"
    abstract="true">
    <property name="country" value="India"></property>
</bean>

<bean id="employeeBean" parent="indianEmployee">
    <property name="employeeId" value="1065"></property>
```

```

        <property name="employeeName" value="Ravi Kant
Soni"></property>
    </bean>
...

```

The parent bean `indianEmployee` cannot be instantiated on its own because it is explicitly marked as `abstract`. When a bean definition is `abstract`, that bean definition is served as a pure template bean definition and used as a parent definition for child definitions. So, while running the `PayrollSystem` class, the following code snippet will result in an error message on the console:

```

...
    // using 'indianEmployee'
    Employee employeeB = (Employee)
context.getBean("indianEmployee");
    System.out.println(employeeB);
...

```

Since the `indianEmployee` bean is a pure template, if you try to instantiate it, you will encounter the following error message:

```

org.springframework.beans.factory.BeanIsAbstractException: Error
creating bean with name 'indianEmployee': Bean definition is
abstract

```

Autowiring in Spring

Setting bean dependencies in the configuration file is a good practice to follow in the Spring Framework; however, the Spring container can automatically autowire relationships between collaborating beans by inspecting the contents of `BeanFactory`.

As we have seen, every member variable in the Spring bean has to be configured; for example, if a bean references another bean, we have to specify the reference explicitly. Autowiring is a feature provided by the Spring Framework that helps us reduce some of these configurations by intelligently guessing what the reference is.

The Spring Framework provides autowiring features where we don't need to provide bean injection details explicitly. The Spring container can autowire relationships between collaborating beans without using the `<constructor-arg>` and `<property>` elements. This immensely helps in cutting down the XML configuration. Spring is capable of automatically resolving dependencies at runtime. This automatic resolution of bean dependencies is also called autowiring.

Spring wires a bean's properties automatically by setting the `autowire` property on each `<bean>` tag that you want to autowire. By default, autowiring is disabled. To enable it, specify the method of autowiring you want to apply using the `autowire` attribute of the bean you want to autowire, as shown here:

```
<bean id="foo" class ="Foo" autowire="autowire-type" />
```

Autowiring modes

There are five modes of autowiring that Spring Container can use for autowiring. They are explained in the following table:

Mode	Description
No	By default, Spring bean autowiring is turned off, that is, no autowiring is to be performed, and you should use explicit bean reference <code>ref</code> for wiring.
byname	This is autowiring by property name, that is, if the bean property is the same as the other bean name, autowire it. The setter method is used for this type of autowiring to inject a dependency.
byType	The data type is used for this type of autowiring. If the data type bean property is compatible with the data type of the other bean, autowire it. Only one bean should be configured for this type in the configuration file; otherwise, a fatal exception is thrown.
constructor	This is similar to autowire <code>byType</code> , but here the constructor is used to inject a dependency.
autodetect	Spring first tries to autowire by the constructor; if it does not work, then Spring tries to autowire with <code>byType</code> . This option is deprecated.

Let's demonstrate autowiring with examples.

In the `EmployeeServiceImpl.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.autowiring;

public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDao employeeDao = null;

    public void setEmployeeDao(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }
}
```

In the `EmployeeDaoImpl.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.autowiring;

public class EmployeeDaoImpl implements EmployeeDao {

    // ...

}
```

In the preceding code snippet, the `EmployeeServiceImpl` class has `employeeDao` field and a setter method.

Autowiring using the no option

This is a default mode, and you should use the explicit bean reference `ref` for wiring.

In the `beans.xml` file, you'll find the following code

```
...
<bean id="employeeService"
      class="org.packt.Spring.chapter2.autowiring.
EmployeeServiceImpl">
    <property name="employeeDao"
ref="employeeDaoBean"></property>
</bean>

<bean id="employeeDaoBean"
      class="org.packt.Spring.chapter2.autowiring.EmployeeDaoImpl">
</bean>
...
```

Autowiring using the byname option

Autowiring using the `byName` option autowires a bean by its property name.

A Spring container looks at the properties of the beans on which the `autowire` attribute is set using `byName` in the configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. If such a bean is found, it is injected into the property. If no such bean is found, an error is raised.

Case 1 – if id=" employeeDao"

In the beans.xml file, you'll find the following code

```
...
    <bean id="employeeService"
        class="org.packt.Spring.chapter2.automwiring.
EmployeeServiceImpl"
        autowire="byName">
    </bean>

    <bean id="employeeDao"
class="org.packt.Spring.chapter2.automwiring.EmployeeDaoImpl">
    </bean>

...
```

In this case, since the name of the employeeDao bean is the same as the employeeService bean's property (EmployeeDao employeeDao), Spring will autowire it via the setter method setEmployeeDao (EmployeeDao employeeDao).

Case 2 – if id=" employeeDaoBean"

In the beans.xml file, you'll find the following code:

```
...
    <bean id="employeeService"
        class="org.packt.Spring.chapter2.automwiring.
EmployeeServiceImpl"
        autowire="byName">
    </bean>

    <bean id="employeeDaoBean" class="org.packt.Spring.chapter2.
automwiring.EmployeeDaoImpl">
    </bean>

...
```

In this case, since the name of the employeeDaoBean bean is not the same as the employeeService bean's property (EmployeeDao employeeDao), Spring will not autowire it via the setter method, setEmployeeDao (EmployeeDao employeeDao). So, the employeeDao property will get a null value.

Autowiring using the byType option

Autowiring using `byType` enables Dependency Injection based on property data types.

The Spring container looks at each property's class type searching for a matching bean definition in the configuration file when autowiring a property to a bean. If no such bean is found, a fatal exception is thrown. If there is more than one bean definition found in the configuration, a fatal exception is thrown, and it will not allow `byType` autowiring for that bean.

If there are no matching beans, nothing happens; the property is not set. So, to throw an error, use the `dependency-check="objects"` attribute value.

In the `beans.xml` file, you'll find the following code

```
...
<bean id="employeeService"
      class="org.packt.Spring.chapter2.autowiring.
EmployeeServiceImpl"
      autowire="byType">
</bean>

<bean id="employeeDaoBean"
      class="org.packt.Spring.chapter2.autowiring.EmployeeDaoImpl">
</bean>
...
```

In this case, since the data type of the `employeeDaoBean` bean is the same as the data type of the `employeeService` bean's property (`EmployeeDao employeeDao`), Spring will autowire it via the setter method `setEmployeeDao(EmployeeDao employeeDao)`.

Autowiring using the constructor

Autowiring using the constructor applies to constructor arguments.

It will look for the class type of constructor arguments and perform autowiring using `byType` on all constructor arguments. A fatal error is raised if there isn't exactly one bean of the constructor argument type in the container.

In the `EmployeeServiceImpl.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.autowiring;

public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDao employeeDao;
```

```
public EmployeeServiceImpl(EmployeeDao employeeDao) {
    this.employeeDao = employeeDao;
}

public EmployeeDao getEmployeeDao() {
    return employeeDao;
}
}
```

In the `beans.xml` file, you'll find the following code

```
...
<bean id="employeeService"
      class="org.packt.Spring.chapter2.autowiring
.EmployeeServiceImpl"
      autowire="constructor">
</bean>

<bean id="employeeDaoBean" class="org.packt.Spring.chapter2.
autowiring.EmployeeDaoImpl">
</bean>
...
```

In this case, since the data type of the `employeeDaoBean` bean is the same as the constructor argument data type in the `employeeService` bean's property (`EmployeeDao employeeDao`), Spring autowires it via the constructor: `public EmployeeServiceImpl(EmployeeDao employeeDao)`.

The bean's scope

Spring provides us with beans after instantiating and configuring them. Spring Container manages objects. This means that any object can refer to any other object from Spring Container using the bean's ID, and Spring Container provides an instance of the requesting object.

When we start Spring Container, `ApplicationContext` reads the Spring configuration file, looks for all bean definitions available there, and then initializes beans before any call to the `getBean()` method.

During initialization, `ApplicationContext` itself has initialized all the Spring beans configured in Spring XML. When another object makes a call to the `getBean()` method, `ApplicationContext` returns the same reference of bean that has already been initialized. This is the default behavior of beans.

This leads to the concept of a bean's scope. We can choose the number of instances of beans depending on the scope. There are different scopes in which a bean can be configured. The `<bean>` tag has a `scope` attribute that is used to configure the scope of the bean. There are different bean scopes in Spring, such as singleton, prototype, request, session, and global session. We will understand each session one by one.

Let's understand this by considering the following example, where we have the `EmployeeService` interface, `EmployeeServiceImpl` class, and `PayrollSystem` class with the `main()` method.

In the `EmployeeService.java` interface, you'll find the following code

```
package org.packt.Spring.chapter2.beanscope;

public interface EmployeeService {
    void setMessage(String message);
    String getMessage();
}
```

In the preceding code snippet, the `EmployeeService` interface declares two methods.

The following are the contents of the `EmployeeServiceImpl.java` class:

```
package org.packt.Spring.chapter2.beanscope;

import org.springframework.beans.factory.InitializingBean;

public class EmployeeServiceImp implements EmployeeService {

    private String message;

    @Override
    public void setMessage(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return this.message;
    }
}
```

In the preceding code snippet, the `EmployeeServiceImpl` class implemented the `EmployeeService` interface.

In the `beans.xml` file, you'll find the following code

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="employeeServiceBean" class="org.packt.Spring.chapter2.
beanscope.EmployeeServiceImpl">
        </bean>

</beans>
```

In the preceding configuration file, we define `employeeServiceBean` without any scope, to see the default nature of the bean.

In the `PayrollSystem.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.beanscope;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
ApplicationContext;

public class PayrollSystem {

    public static void main(String[] args) {

        ApplicationContext context = new
ClassPathXmlApplicationContext(
            "beans.xml");

        // Retrieve for first time
        EmployeeService employeeServiceA = (EmployeeService)
context
            .getBean("employeeServiceBean");
        employeeServiceA.setMessage("Message by service A");

        System.out
            .println("employeeServiceA: " +
employeeServiceA.getMessage());
    }
}
```

```

        // Retrieve it again
        EmployeeService employeeServiceB = (EmployeeService)
context
        .getBean("employeeServiceBean");
        System.out
        .println("employeeServiceB: " +
employeeServiceB.getMessage());
    }
}

```

In the preceding code snippet, the `PayrollSystem` class has the `main()` method. For the first time, we call `getBean("employeeServiceBean")`, assign the bean to the `employeeServiceA` variable of the `EmployeeService` type, and then set the message by calling the `setMessage()` method. Again, we call `getBean("employeeServiceBean")` and assign the bean to the `employeeServiceB` variable of the `EmployeeService` type. The output after calling the `getMessage()` method from both reference variable results is the same, as shown here:

```

org.springframework.context.support.ClassPathXmlApplicationContext
prepareRefresh
INFO: Refreshing org.springframework.context.support.
ClassPathXmlApplicationContext
@1202d69: startup date [Sat Jan 24 20:04:30 IST 2015]; root of
context hierarchy
Jan 24, 2015 8:04:30 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[beans.xml]
employeeServiceA: Message by service A
employeeServiceB: Message by service A

```

Singleton

By default, all Spring beans are singleton. Once `ApplicationContext` is initialized, it looks at all the beans in XML and initializes only one bean per bean definition in Spring Container. On each call to the `getBean()` method, Spring Container returns the same instance of the bean.

The first bean scope in Spring that is called is singleton, which initializes only one bean per bean definition in the container and returns the same instance reference on each call to the `getBean()` method. This scope makes Spring initialize all beans during the load time itself without waiting for the `getBean()` call.

In the `beans.xml` file, you'll find the following code

```
...
<bean id="employeeServiceBean" class="org.packt.Spring.chapter2.
beanscope.EmployeeServiceImpl"
      scope="singleton">
</bean>
...
```

In the preceding configuration file, we have a bean with a singleton scope. When we run `PayrollSystem.java`, the output will be as follows:

```
org.springframework.context.support.ClassPathXmlApplicationContext
prepareRefresh
INFO: Refreshing org.springframework.context.support.
ClassPathXmlApplicationContext
@1855562: startup date [Sat Jan 24 20:36:27 IST 2015]; root of
context hierarchy
Jan 24, 2015 8:36:28 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[beans.xml]
employeeServiceA: Message by service A
employeeServiceB: Message by service A
```

Since the `EmployeeServiceImpl` bean is in the singleton scope, the second retrieval by `employeeServiceB` will display the message set by `employeeServiceA` even though it's retrieved by calling a new `getBean()` method.

The singleton pattern in general says that overall there will be only one instance of the object. But when we talk about singleton in the Spring Framework, we are talking about Spring Container alone.

We can have multiple containers running in the same JVM, so we can have multiple instances of the same bean in same JVM.

So, singleton in Spring represents in a particular Spring container, and there is only one instance of a bean created in that container that is used across different references.

Prototype

The prototype is second bean scope in Spring, which returns a brand-new instance of a bean on each call to the `getBean()` method. When a bean is defined as a prototype, Spring waits for `getBean()` to happen and only then does it initialize the prototype. For every `getBean()` call, Spring has to perform initialization, so instead of doing default initialization while a context is being created, it waits for a `getBean()` call. So, every time `getBean()` gets called, it creates a new instance.

In the `beans.xml` file, you'll find the following code

```
...
    <bean id="employeeServiceBean" class="org.packtd.Spring.chapter2.
beanscope.EmployeeServiceImpl"
        scope="prototype">
    </bean>
...
```

In the preceding configuration file, we have a bean with scope as a prototype. When we run the `PayrollSystem.java` file, the output will be as follows

```
org.springframework.context.support.ClassPathXmlApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext
@1855562: startup date [Sat Jan 24 21:05:14 IST 2015]; root of
context hierarchy
Jan 24, 2015 9:05:15 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[beans.xml]
employeeServiceA: Message by service A
employeeServiceB: null
```

The configured destruction life cycle callbacks are not called in the case of a prototype. Spring doesn't maintain the complete life cycle of the prototype. Here, the container instantiates and configures prototype beans and returns this bean to the client with no further record of this prototype instance.

Since every `getBean()` call creates a new instance of the prototype bean, this could lead to performance issues when beans use limited resources such as network connections, whereas it may be useful if you would like to get a new instance of a domain object, such as an `employee` object.

Request

The third bean scope in Spring is request, which is available only in web applications that use Spring and create an instance of bean for every HTTP request. Here, a new bean is created per Servlet request. Spring will be aware of when a new request is happening because it ties well with the Servlet APIs, and depending on the request, Spring creates a new bean. So, if the request scope has `getBean()` inside it, for every new request, there will be a new bean. However, as long as it's in the same request scope, the same bean is going to be used.

Session

The session is the fourth bean scope in Spring, which is available only in web applications that use Spring and create an instance of bean for every HTTP session. Here, a new bean is created per session. As long as there is one user accessing in a single session, each call to `getBean()` will return same instance of the bean. But if it's a new user in a different session, then a new bean instance is created.

Global session

The global session is the fifth bean scope in Spring, which works only in portlet environments that use Spring and create a bean for every new portlet session.

The Spring bean life cycle

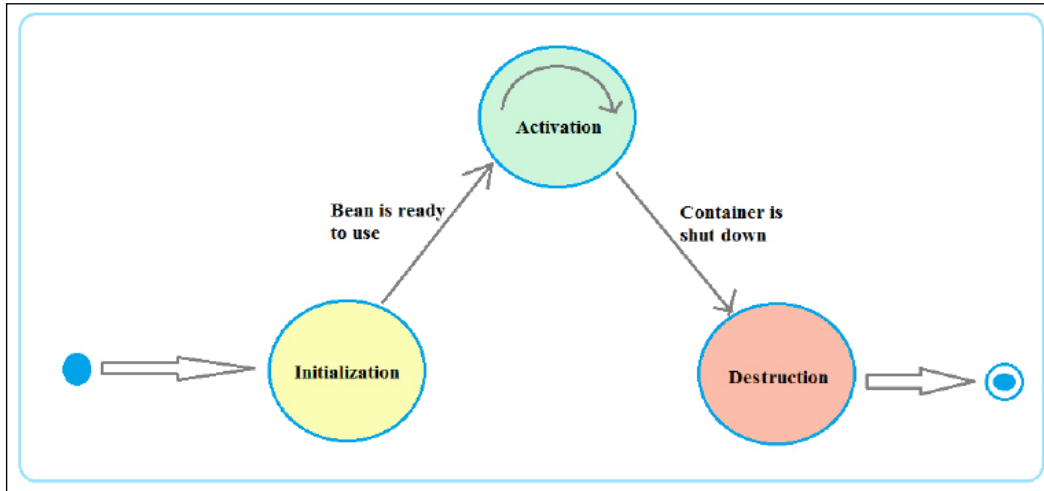
As long as Spring beans are required by the application, they exist within the container. For a bean to get into a usable state after instantiation, it needs to perform some initialization. Likewise, some clean up may be necessary when the bean is no longer required and is removed from the container.

Spring provides us with callback methods for the life cycle of the bean. You can have a method in your bean that runs when the bean has been created, and you can also have a method in your bean that is run when the bean is about to be destroyed.

Spring's `BeanFactory` manages the life cycle of beans created through the Spring IoC container. The life cycle of beans consist of callback methods, which can be categorized broadly into the following two groups:

- Post-initialization callback methods
- Pre-destruction callback methods

The following figure illustrates the two groups



Initialization

It represents a sequence of activities that take place between the bean instantiation and the handover of its reference to the client application:

- The bean container finds the definition of the Spring bean in the configuration file and creates an instance of the bean
- If any properties are mentioned, populate the properties using setters
- If the Bean class implements the `BeanNameAware` interface, then call the `setBeanName()` method
- If the Bean class implements the `BeanFactoryAware` interface, then call the `setBeanFactory()` method
- If the Bean class implements the `ApplicationContextAware` interface, then call the `setApplicationContext()` method
- If there are any `BeanPostProcessors` objects associated with the `BeanFactory` interface that loaded the bean, then Spring will call the `postProcessBeforeInitialization()` method before the properties for the bean are injected
- If the Bean class implements the `InitializingBean` interface, then call the `afterPropertiesSet()` method once all the bean properties defined in the configuration file are injected

- If the bean definition in the configuration file contains the `init-method` attribute, then call this method after resolving the value for the attribute to a method name in the Bean class
- The `postProcessAfterInitialization()` method will be called if there are any bean post processors attached to the `BeanFactory` interface that loads the bean

Activation

The bean has been initialized and the dependency has been injected. Now the bean is ready to be used by the application.

Destruction

This represents the following sequence of activities:

- If the Bean class implements the `DisposableBean` interface, then call the `destroy()` method when the application no longer needs the bean reference
- If the bean definition in the configuration file contains the `destroy-method` attribute, then call this method after resolving the value for the attribute to a method name in the Bean class.

There are two important bean life cycle callback methods that are required at the time of bean initialization and its destruction.

- Initialization callbacks
- Destruction callbacks

Initialization callbacks

There are two ways in which you can achieve the initialization work after all necessary properties on the bean are set by the container:

- Implementing the `org.springframework.beans.factory.InitializingBean` interface
- Using `init-method` in the XML configuration

In the `EmployeeService.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.callbacks;

public interface EmployeeService {
```

```

    public Long generateEmployeeID();

}

```

Implementing the org.springframework.beans.factory.InitializingBean interface

The `org.springframework.beans.factory.InitializingBean` interface is used to specify a single method in a bean, as follows:

```
void afterPropertiesSet() throws Exception;
```

This method gets initialized whenever the bean containing this method is called.

In the `EmployeeServiceImpl.java` class, you'll find the following code

```

package org.packt.Spring.chapter2.callbacks;

import org.springframework.beans.factory.InitializingBean;

public class EmployeeServiceImpl implements EmployeeService,
    InitializingBean {

    @Override
    public Long generateEmployeeID() {

        return System.currentTimeMillis();

    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("Employee afterPropertiesSet... ");
    }

}

```

In the `beans.xml` file, you'll find the following code

```

...
<bean id="employeeServiceBean"
class="org.packt.Spring.chapter2.callbacks.EmployeeServiceImpl">
</bean>
...

```

Here, the `InitializingBean` interface tells Spring that the `EmployeeServiceImpl` bean needs to know when it's being initialized. A method of this bean needs to be called when the bean is initialized. The `InitializingBean` interface has `afterPropertiesSet()`, which needs to be implemented, and it will be called by Spring when this bean is initialized and all properties are set. This `InitializingBean` interface is a marker for the bean to know that the `afterPropertiesSet()` method of this bean needs to be called after initialization.

Using init-method in the XML configuration

In the case of XML-based configuration metadata, you can use the `init-method` attribute to specify the name of the method that has a void no-argument signature, which is to be called on the bean immediately upon instantiation.

In the `beans.xml` file, you'll find the following code

```
...
<bean id="employeeServiceBean" class="org.packt.Spring.chapter2.
callbacks.xml.EmployeeServiceImpl"
init-method="myInit">
</bean>
...
```

In the `EmployeeServiceImpl.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.callbacks.xml;

public class EmployeeServiceImpl implements EmployeeService {

    @Override
    public Long generateEmployeeID() {
        return System.currentTimeMillis();
    }

    public void myInit() {
        System.out.println("Employee myInit... ");
    }
}
```

Now we have `init-method` in the configuration `beans.xml` file, which will take the method name as the value from the bean. So, instead of implementing an interface to this bean, we have a simple method that is called by Spring.

Destruction callbacks

There are two ways you can do a destruction callback:

- Implementing the `org.springframework.beans.factory.DisposableBean` interface
- Using `destroy-method` in the XML configuration

Implementing the `org.springframework.beans.factory.DisposableBean` interface

The `org.springframework.beans.factory.DisposableBean` interface is used to specify a single method in a bean, as follows:

```
void destroy() throws Exception;
```

This method allows a bean to get a callback whenever the Spring container containing this bean is destroyed.

In the `EmployeeServiceImp.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.callbacks;

import org.springframework.beans.factory.DisposableBean;

public class EmployeeServiceImp implements EmployeeService,
DisposableBean {

    @Override    public Long generateEmployeeID() {
        return System.currentTimeMillis();
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("Employee destroy... ");
    }
}
```


In the `beans.xml` file, you'll find the following code

```
...
<bean id="employeeServiceBean" class="org.packt.Spring.chapter2.
callbacks.EmployeeServiceImpl">
</bean>
...
```

The `DisposableBean` interface has a `destroy()` method. If a bean implements a `DisposableBean` interface, then Spring will automatically call the `destroy()` method of that bean before actually destroying the bean.

Using destroy-method in the XML configuration

In the case of XML-based configuration metadata, you can use the `destroy-method` attribute to specify the name of the method that has a void no-argument signature, which is called just before a bean is removed from the container.

In the `beans.xml` file, you'll find the following code

```
<bean id="employeeServiceBean" class="org.packt.Spring.chapter2.
callbacks.xml.EmployeeServiceImpl"
destroy-method="cleanUp">
</bean>
```

In the `EmployeeServiceImpl.java` class, you'll find the following code:

```
package org.packt.Spring.chapter2.callbacks.xml;

public class EmployeeServiceImpl implements EmployeeService {

    @Override
    public Long generateEmployeeID() {
        return System.currentTimeMillis();
    }

    public void cleanUp() {
        System.out.println("Employee Cleanup... ");
    }
}
```

Now we have `destroy`-method in the configuration `beans.xml` file, which will take the method name as a value from the bean. So, instead of implementing the interface to this bean, we have a simple method that is called by Spring.

In the `PayrollSystem.java` class, you'll find the following code

```
package org.packt.Spring.chapter2.callbacks.xml;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class PayrollSystem {

    public static void main(String[] args) {
        ConfigurableApplicationContext context = new
        ClassPathXmlApplicationContext("beans.xml");
        EmployeeService employeeService = (EmployeeService)
        context.getBean("employeeServiceBean");
        System.out.println(employeeService.generateEmployeeID());
        context.close();
    }
}
```

Exercise

- Q1. What are Inversion of Control (IoC) and Dependency Injection (DI)?
- Q2. What are the different types of Dependency Injection in Spring?
- Q3. Explain autowiring in Spring. What are the different modes of autowiring.
- Q4. Explain the different Spring bean scopes.



The answers to these are provided in *Appendix A, Solution to Exercises*.

Summary

In this chapter, you learned about the Spring IoC container and the `BeanFactory` and `ApplicationContext` interfaces. You also learned about DI in Spring and their types. We saw the bean's scope in Spring. Finally, we went through the life cycle of the Spring bean.

In the next chapter, we will cover the DAO design pattern. We will take a look at a simplified spring JDBC abstraction framework. We will implement the JDBC code using the Spring JDBC support and discuss how Spring manages `DataSource` and the data sources you can use in your applications. We will also discuss data support in Spring applications.

3

DAO and JDBC in Spring

In the previous chapter, you explored the concept of **Inversion of Control (IoC)**. We then explored concepts such as, `Spring Core Container`, `BeanFactory`, and `ApplicationContext`, and then you learned how to implement them. We looked at **Dependency Injection (DI)** in Spring and their types: setter and constructor. We also wired beans using setter- and constructor-based Dependency Injection for the different data types.

In this chapter, we will cover the **Data Access Object (DAO)** design pattern. We will look at the simplified spring JDBC abstraction framework. We will implement the JDBC code using the Spring JDBC support and discuss how Spring manages `DataSource` that you can use in your applications. We will discuss data support in the Spring application.

When we talk about the Spring data support, it's specifically for the purpose of your application interacting with the data or the database, and you can typically write the Java code that interacts with the database. There are a few things that you have to do irrespective of what code you are going to write. You need to open the connection, manage the transaction, and then close the connection to write some boilerplate code. The whole point of using the Spring data support is that you can do away with all the extra boilerplate code and the code that you write specifically for the business case and the business problem that you want to resolve.

When we talk about writing a code that interacts with the database in Java, there are numerous ways we can do that. It could be as simple as JDBC or it could be some kind of framework, such as **Hibernate** or **iBATIS**. Spring supports lots of these technologies. The Spring JDBC module provides a kind of an abstraction layer and all the tedious JDBC code that we would otherwise have to write is provided by the JDBC module, which is in the Spring Framework.

The topics covered in this chapter are listed as follows:

- Overview of database
- The DAO design pattern
- JDBC without Spring
- The Spring JDBC packages
- JDBC with Spring
- What is `JdbcTemplate`
- The JDBC batch operation in Spring
- Calling the stored procedure

Overview of database

Databases are everywhere, but you never see them. They are concealed behind the tools and services that you use every day. Ever wondered where Facebook, Twitter, and Tumbler store their data? The answer is a database. Where does Google keep the details of the pages that it indexes from the Internet and where are the contacts stored in your mobile phone? Again, the answer is a database. In the information system, databases do most of the work that we do in our day-to-day lives. So, what is a database?

A database is a place where we store data. Databases are organized and structured. All the data that we store in the database fits into the database structure. Flat-file databases are simple databases. They store data in columns and rows.

Let's look at an Employee table:

Employee ID	First name	Last name	Age	Contact number
1	Ravi	Soni	28	+91-9986XXXXXX
2	Shree	Kant	22	+91-9986XXXXXX

Let's think about a simple database that the **Human Resource (HR)** has used to store his/her employee details. This database contains the name, address, birth date, and contact number of each employee. If the HR hires a new employee and would like to add the employee details to the database, then the HR will store the employee's first name, last name, address, date of birth, and mobile number in the database. The employee details that the HR writes down are stored in the fields of his employee address database. Each row is called a record, and each of the rows holds the information about the different employees in his/her employee address database. So, unlike a paper employee address book, the HR can carry out employee-related operations on his/her stored database. They can use the search option to find a particular employee's details.

In almost any business these days, there is a database or a collection of databases, and these are the main pieces of the backend infrastructure. Database is nothing but collection of data. There are different kinds of databases, such as Oracle, PostgreSQL, MySQL, and so on. The database software is called a **relational database management system (RDBMS)** and its instance is called a database engine. The database server is a machine that runs the database engine. We refer to the RDBMS, when we mention the term database throughout this book.



Refer to *Appendix B, Apache Derby Database*, to set up the Apache Derby database.

The DAO design pattern

The DAO design pattern can be used to provide a separation between the low-level data accessing operations and the high-level business services, as shown here:



The DAO layer

In between the database and the business layer, there is a layer called the DAO layer. The DAO layer is mainly used to perform the **Create-Retrieve-Update-Delete (CRUD)** operation. The DAO layer is responsible for creating, obtaining, updating, or deleting records in the database table. To perform this CRUD operation, DAO uses a low-level API, such as the JDBC API or the Hibernate API. This DAO layer will have a method for performing the CRUD operation. It is the intermediate layer between the **Business Layer** and the **DB**. It is used to separate the low-level accessing API from the high-level business service. The DAO layer decouples the implementation of persistent storage from the rest of your application.

The advantages of using DAO are as follows:

- Its application is independent of the data access techniques and database dependency
- It offers loose coupling with the other layers of the application
- It helps the unit test the service layer using a mock object without connecting to the database

JDBC without Spring

As Java developers, we work with data all the time and develop almost all the applications that interact with some sort of database and most of the times it's relational. Generally, the application needs to interact with a database in order to get data from it. And the typical way for connecting a Java application to a database would be through JDBC.

Java Database Connectivity (JDBC) is a standard Java API. It is used for database connectivity between the Java programming language and a great variety of databases. JDBC is an application programming interface that allows a Java programmer to access the database from a Java code using sets of standard interfaces and classes written in a Java programming language.

JDBC provides several methods for querying, updating, and deleting data in RDBMS, such as SQL, Oracle, and so on. The JDBC library provides APIs for tasks such as:

- Making a connection to a database
- Creating the SQL statements
- Executing the SQL queries in the database

- Viewing and modifying the resulting records
- Closing a database connection

It is generally considered a pain to write a code to get JDBC to work. We need to write a boilerplate code to open a connection and to handle the data. Another problem with JDBC is that of poor exception hierarchy, such as `SQLException`, `DataTruncation`, `SQLWarning`, and `BatchUpdateException`. These require less explanation and a major problem is that all of these exceptions are deployed as checked exceptions, which mandate the developer to go ahead to implement a try block. It's very difficult to recover from a catch block, when an exception is thrown even during the statement execution, and most of the time these catch blocks are used for generating log messages for those exceptions.

Sample code

Here, we will take the example of `JdbcHrPayrollSystem`, which connects to the Apache Derby database that we saw in the previous section. We will write a query to retrieve the record, we will look at the code required to run this query, and then we will print out the retrieved record.

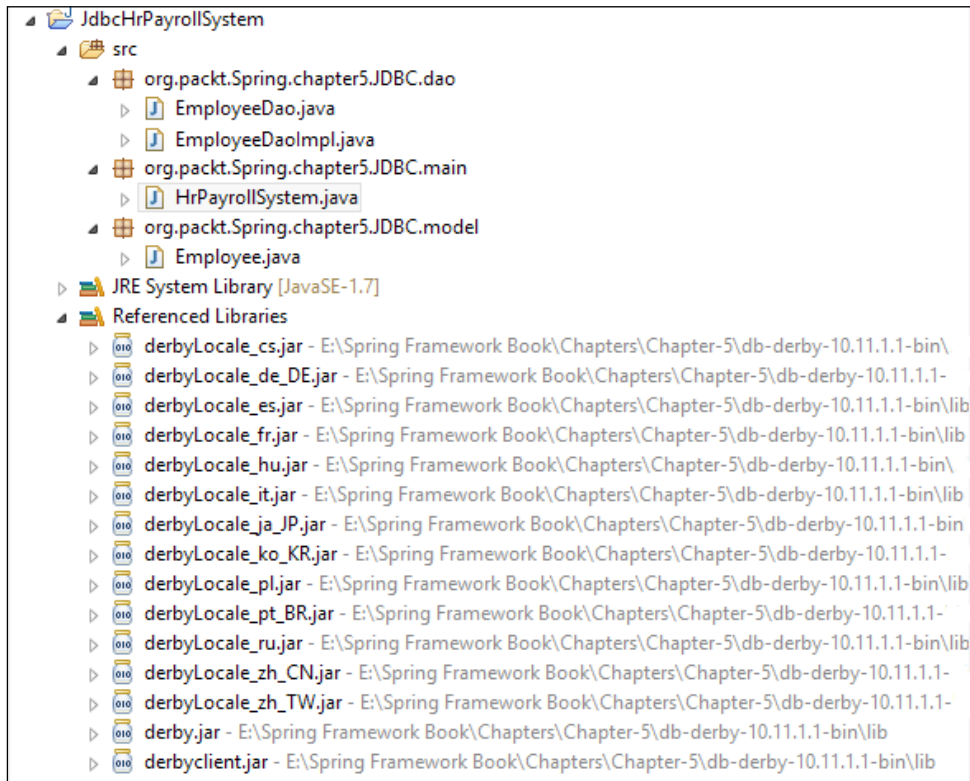
ADD drivers specific to database into the project

Whenever we need to write a code to access the database, we have to make sure that the drivers for the database that we are trying to connect to are available for the project. For Apache Derby, we need to include a driver so that the project can connect to the database, as shown here:

```
project > properties > Libraries > Add External jars > (navigate  
to the derby folder) > lib folder > select (derby.jar and  
derbyclient.jar) > ok
```


Directory structure of the application

The final directory structure of the application is shown in the following screenshot:



It is a good practice to design DAO using the **program to an interface** principle, which states that concrete implementations must implement the interface that is used in the program that wants to use the implementation rather than the implementation class itself. Following this principle, we will first define an interface for `EmployeeDao` and declare some data access methods that include the methods for creating new employee details, or getting employee details using the employee ID, and then inserting the employee details into the table.

The Employee.java file

We have package `org.packt.Spring.chapter5.JDBC.model` that contains the class named `employee`, which is a simple model class containing the employee ID, name, and its corresponding getter and setter. This `employee` class also has a parameterized constructor with parameters, such as `id` and `name` that set the instance variable:

```
package org.packt.Spring.chapter5.JDBC.model;

public class Employee {

    private int id;
    private String name;

    public Employee(int id, String name) {
        setId(id);
        setName(name);
    }

    // setter and getter
}
```

The EmployeeDao.java file

We have package `org.packt.Spring.chapter5.JDBC.dao` that has the interface `EmployeeDao` and the class `EmployeeDaoImp`. This interface contains the method for creating the `Employee` table, inserting the values into the table, and fetching the employee data from the table based on the employee ID, as shown here:

```
package org.packt.Spring.chapter5.JDBC.dao;

import org.packt.Spring.chapter5.JDBC.model.Employee;

public interface EmployeeDao {
    // get employee data based on employee id
    Employee getEmployeeById(int id);
    // create employee table
    void createEmployee();
    // insert values to employee table
    void insertEmployee(Employee employee);
}
```

The EmployeeDaoImpl.java file

Now, we will provide an implementation for the `EmployeeDao` interface. The `EmployeeDaoImpl` class is responsible for connecting to the database and getting or setting the values. The complexity lies in the JDBC code that goes inside the methods that connect to the database. First, we need to have a connection object, and then we need to initialize `ClientDriver`, which in our case, is specific to the Apache Derby driver. Now, we need to open a connection using the database URL. Then, based on the functionality, we need to prepare and execute a query:

```
package org.packt.Spring.chapter5.JDBC.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import org.packt.Spring.chapter5.JDBC.model.Employee;

public class EmployeeDaoImpl implements EmployeeDao {
    // JDBC driver name and database URL
    static final String JDBC_DRIVER =
"org.apache.derby.jdbc.ClientDriver";
    static final String DB_URL = "jdbc:derby://localhost:1527/db";

    private void registerDriver() {
        try {
            Class.forName(JDBC_DRIVER).newInstance();
        } catch (InstantiationException e) {
        } catch (IllegalAccessException e) {
        } catch (ClassNotFoundException e) {
        }
    }
}
```

Here, the `getEmployeeById(int id)` method will fetch the employee information based on the employee ID:

```
@Override
public Employee getEmployeeById(int id) {
    Connection conn = null;
    Employee employee = null;
```

```

    try {
        // register apache derby driver
        registerDriver();
        // open a connection using DB url
        conn = DriverManager.getConnection(DB_URL);
        // Creates a PreparedStatement object for sending
parameterized SQL
        // statements to the database
        PreparedStatement ps = conn
            .prepareStatement("select * from
employee where id = ?");
        // Sets the designated parameter to the given Java
int value
        ps.setInt(1, id);
        // Executes the SQL query in this PreparedStatement
object and
        // returns the ResultSet object
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            employee = new Employee(id,
rs.getString("name"));
        }
        rs.close();
        ps.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
            }
        }
    }
    return employee;
}

```

The `createEmployee()` method creates an `Employee` table with the column `ID` and `name`, as shown in the following code snippet:

```
@Override
public void createEmployee() {
    Connection conn = null;
    try {
        // register apache derby driver
        registerDriver();
        // open a connection using DB url
        conn = DriverManager.getConnection(DB_URL);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("create table employee (id
integer, name char(30))");
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
            }
        }
    }
}
```

In the following code snippet, the `insertEmployee(Employee employee)` method will insert the employee information into the `Employee` table:

```
@Override
public void insertEmployee(Employee employee) {
    Connection conn = null;
    try {
        // register apache derby driver
        registerDriver();
        // open a connection using DB url
        conn = DriverManager.getConnection(DB_URL);
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("insert into employee values ("
            + employee.getId() + ", ' " +
employee.getName() + "')");
        stmt.close();
    } catch (SQLException e) {
```

```

        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
            }
        }
    }
}

```

The HrPayrollSystem.java file

We have package `org.packt.Spring.chapter5.JDBC.main` that contains the class `HrPayrollSystem` with the `main()` method. In the `main()` method, we will initialize DAO and call the methods of DAO to create a table, insert the data, and then fetch the data from the table, as shown here:

```

package org.packt.Spring.chapter5.JDBC.main;

import org.packt.Spring.chapter5.JDBC.dao.EmployeeDao;
import org.packt.Spring.chapter5.JDBC.dao.EmployeeDaoImpl;
import org.packt.Spring.chapter5.JDBC.model.Employee;

public class HrPayrollSystem {

    public static void main(String[] args) {
        EmployeeDao employeeDao = new EmployeeDaoImpl();
        // create employee table
        employeeDao.createEmployee();
        // insert into employee table
        employeeDao.insertEmployee(new Employee(1, "Ravi"));
        // get employee based on id
        Employee employee = employeeDao.getEmployeeById(1);
        System.out.println("Employee name: " +
employee.getName());
    }
}

```

Having shown the trouble in using JDBC, in the next section, we will be discussing the DAO support in the Spring Framework to remove the troubling points one after the other.

Spring JDBC packages

In the previous section, we have seen the shortcomings of using the JDBC API as a low-level data access API for implementing the DAOs. These shortcomings are as follows:

- **Code duplication:** As we know, writing the boilerplate code over and over again in code duplication violates the **Don't repeat yourself (DRY)** principle. This has some side effects in terms of the project costs, efforts, and timelines.
- **Resource leakage:** The DAO methods must hand over the control of the obtained database resources, such as connection, statements, or result sets after calling the `close()` method. This is a risky plan because a novice programmer might very easily skip some of the code fragments. As a result, the resources would run out and bring the system to a stop.
- **Error handling:** When using JDBC directly we need to handle `SQLException`, since the JDBC drivers report all the errors suitable by raising `SQLException`. It is not possible to recover these exceptions. Moreover, the message and the error code obtained from the `SQLException` object are database vendor-specific, so it is difficult to write a portable DAO error messaging code.

To solve the aforementioned problems, we need to identify the parts of the code that are fixed and then encapsulate them into some reusable objects. The Spring Framework provides a solution for these problems by giving a thin, robust, and highly extensible JDBC abstraction framework.

The JDBC abstraction framework provided under the Spring Framework is considered to be a value-added service that takes care of all the low-level details, such as retrieving connection, preparing the statement object, executing the query, and releasing the database resources. While using it for data access, the application developer needs to specify the SQL statement for executing and retrieving the result.

To handle the different aspects of JDBC, Spring JDBC is divided into packages, as shown in the following table:

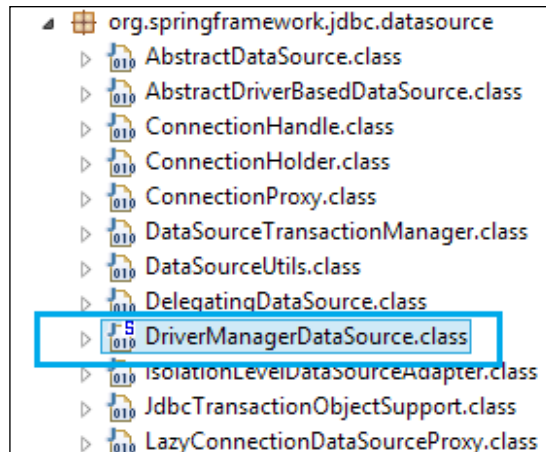
Spring JDBC package	Description
<code>org.springframework.jdbc.core</code>	In the Spring Framework, this package contains the foundation of the JDBC classes, which includes the core JDBC class and <code>JdbcTemplate</code> . It simplifies the database operation using JDBC.
<code>org.springframework.jdbc.datasource</code>	This package contains <code>DataSource</code> implementations and helper classes, which can be used to run the JDBC code outside the JEE container.
<code>org.springframework.jdbc.object</code>	In the Spring Framework, this package contains the classes that help in converting the data returned from the database into plain Java objects.
<code>org.springframework.jdbc.support</code>	<code>SQLExceptionTranslator</code> is the most important class in this package of the Spring Framework. The Spring Framework recognizes the error code used by the database. This is done by using this class and mapping the error code to a higher level of exception.
<code>org.springframework.jdbc.config</code>	This package contains the classes that support JDBC configuration within <code>ApplicationContext</code> of the Spring Framework.

JDBC with Spring

In the earlier section, we did not include any Spring-related functionality, and we implemented a Java class that had DAO implementation, which connected to a database to fetch a particular record using JDBC. Now in this section, we will look at some of the features of the Spring Framework that make our job easier by eliminating the boilerplate code. Here, we will look into the connection support provided by Spring that makes it easy to handle the connections.

DataSource

The `DriverManagerDataSource` class is used for configuring the `DataSource` for application, which is defined in the configuration file, that is `Spring.xml`. So, first of all, we need to add the Spring JAR that will have the `DriverManagerDataSource` class to our project. The Spring Framework provides the JAR for JDBC `spring-jdbc-4.1.4.RELEASE.jar` containing the package named `DataSource`, which will have the class `DriverManagerDataSource.class`, as shown in the following screenshot:



The configuration of `DriverManagerDataSource` is shown here. We need to provide the driver class name and the connection URL. We can also add the username and the password in the property if the database requires it.

Check out the file `Spring.xml` using the following code snippet:

```
...
<context:annotation-config />

<context:component-scan base-
package="org.packt.Spring.chapter5.JDBC.dao" />

<bean id="dataSource"
class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
    <property name="driverClassName"
value="${jdbc.driverClassName}" />
```

```

        <property name="url" value="\${jdbc.url}" />
    </bean>

    <context:property-placeholder location="jdbc.properties" />
    ...

```

The bold properties in the aforementioned configuration code represent the values that you normally pass to JDBC to connect it with the interface. For easy substitution in the different deployment environments and for easy maintenance, the database connection information is stored in the properties file, and the Spring's property placeholder will load the connection information from the `jdbc.properties` file

```

jdbc.driverClassName=org.apache.derby.jdbc.ClientDriver
jdbc.url=jdbc:derby://localhost:1527/db

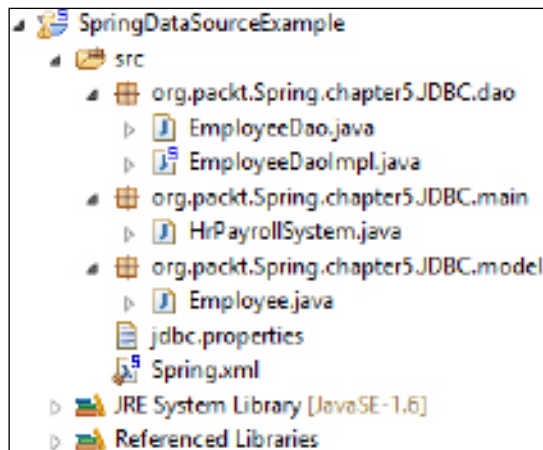
```

DataSource in the DAO class

In the previous section, we added the properties for the `DataSource` in the configuration file `Spring.xml`. So, we will look into the DAOs class to see the benefit of using `DataSource`. We will implement the `EmployeeDao` interface that we defined in the earlier section.

Directory structure of the application

The final directory structure of the application is shown in the following screenshot:



The EmployeeDaoImpl.java file

In the earlier section, we were trying to perform a few basic steps, which are common for methods such as:

- Set up connection to a database
- Create a prepared statement

The first step is to connect to the database that is common for all the methods of the application. We will take out the boilerplate code for this step from the methods defined in the `EmployeeDaoImpl` class.

We have defined `DataSource` as a member variable and annotated it by the `@Autowired` annotation. We have called the `getConnection()` method of this `DataSource` to get the connection based on the definition provided in the configuration file

Checkout the file `EmployeeDaoImpl.java` for the following code snippet:

```
package org.packt.Spring.chapter5.JDBC.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.DataSource;
import org.packt.Spring.chapter5.JDBC.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class EmployeeDaoImpl implements EmployeeDao {
    @Autowired
    private DataSource dataSource;
```

Here, the `EmployeeDaoImpl` class is annotated by the stereotypical annotation, `@Repository`, so that Spring automatically scans this class and registers it as the Spring bean `employeeDaoImpl`.

The `getEmployeeById(int id)` method is used to get the employee details based on the employee ID, as shown here:

```
@Override
public Employee getEmployeeById(int id) {
    Employee employee = null;
```

```

        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn
                .prepareStatement("select * from
employee where id = ?");
            ps.setInt(1, id);
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                employee = new Employee(id,
rs.getString("name"));
            }
            rs.close();
            ps.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }
        return employee;
    }
}

```

The `createEmployee()` method is used for creating the Employee table, as shown in the following code snippet:

```

@Override
public void createEmployee() {
    Connection conn = null;
    try {
        conn = dataSource.getConnection();
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("create table employee (id
integer, name char(30))");
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {

```

```
        conn.close();
    } catch (SQLException e) {
    }
}
}
```

The `insertEmployee(Employee employee)` method is used for inserting the data into the `Employee` table, as shown here:

```
@Override
public void insertEmployee(Employee employee) {
    Connection conn = null;
    try {
        conn = dataSource.getConnection();
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("insert into employee values ("
            + employee.getId() + ", '" +
employee.getName() + "')");
        stmt.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
            }
        }
    }
}
```

The `HrPayrollSystem.java` file

We have package `org.packt.Spring.chapter5.JDBC.main` that contains the class `HrPayrollSystem` with the `main()` method:

```
package org.packt.Spring.chapter5.JDBC.main;

import org.packt.Spring.chapter5.JDBC.dao.EmployeeDao;
import org.packt.Spring.chapter5.JDBC.model.Employee;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;
```

```

public class HrPayrollSystem {

    public static void main(String[] args) {
        @SuppressWarnings("resource")
        ApplicationContext context = new
        ClassPathXmlApplicationContext(
            "Spring.xml");
        EmployeeDao employeeDao =
        context.getBean("employeeDaoImpl",
            EmployeeDao.class);
        // create employee table
        employeeDao.createEmployee();
        // insert into employee table
        employeeDao.insertEmployee(new Employee(1, "Ravi"));
        // get employee based on id
        Employee employee = employeeDao.getEmployeeById(1);
        System.out.println("Employee name: " +
        employee.getName());
    }
}

```

The types of code that we have discussed so far use the Spring Framework to manage `DataSource` and this makes things simple. We have taken all the connection parameters from the class and set them to bean defined by an XML file. In DAO, we have used the method of the new bean to get the connection of the database.

What is JdbcTemplate

The central class of the Spring JDBC abstraction framework is the `JdbcTemplate` class that includes the most common logic in using the JDBC API to access data, such as handling the creation of connection, statement creation, statement execution, and release of resource. The `JdbcTemplate` class can be found in the `org.springframework.jdbc.core` package.

The `JdbcTemplate` class instances are thread-safe once configured. A single `JdbcTemplate` can be configured and injected into multiple DAOs

We can use the `JdbcTemplate` to execute the different types of SQL statements. **Data Manipulation Language (DML)** is used for inserting, retrieving, updating, and deleting the data in the database. `SELECT`, `INSERT`, or `UPDATE` statements are examples of DML. **Data Definition Language (DDL)** is used for either creating or modifying the structure of the database objects in the database. `CREATE`, `ALTER`, and `DROP` statements are examples of DDL.

The `JdbcTemplate` class is in the `org.springframework.jdbc.core` package. It is a non-abstract class. It can be initiated using any of the following constructors:

- `JdbcTemplate`: Construct a new `JdbcTemplate` object. When constructing an object using this constructor, we need to use the `setDataSource()` method to set the `DataSource` before using this object for executing the statement.
- `JdbcTemplate(DataSource)`: Construct a new `JdbcTemplate` object, and initialize it with a given `DataSource` to obtain the connections for executing the requested statements.
- `JdbcTemplate(DataSource, Boolean)`: Construct a new `JdbcTemplate` object, and initialize it by a given `DataSource` to obtain the connections for executing the requested statements, and the `Boolean` value describing the lazy initialization of the SQL exception translator.

If the `Boolean` argument value is `true`, then the exception translator will not be initialized immediately. Instead, it will wait until the `JdbcTemplate` object is used for executing the statement. If the `Boolean` argument value is `false`, then the exception translator will be initialized while constructing the `JdbcTemplate` object.

It also catches the JDBC exception and translates it into the generic and more informatics exception hierarchy, which is defined in the `org.springframework.dao` package. This class avoids common error and executes the SQL queries, updates the statements, stores the procedure calls, or extracts the results.

While using the `JdbcTemplate`, the application developer has to provide the code for preparing the SQL statement and the extract result. In this section, we will look into operations such as, query, update, and so on using the `JdbcTemplate` in Spring.

Configuring the JdbcTemplate object as Spring bean

The Spring `JdbcTemplate` makes the application developer's life a lot easier by taking care of all the boilerplate code required for creating and releasing database connection, which saves development time. In the earlier section, we saw how to define the `DataSource` bean in the configuration file. To initialize the `JdbcTemplate` object, we will use the `DataSource` bean as `ref`. This is discussed while explaining the configuration file `Spring.xml`.

The Spring.xml file

The following code snippet shows the Spring.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
3.2.xsd
  http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">

  <context:annotation-config />

  <context:component-scan base-
package="org.packt.Spring.chapter5.JDBC.dao" />

  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
    <property name="driverClassName">
      <value={jdbc.driverClassName}></value>
    </property>
    <property name="url">
      <value={jdbc.url}></value>
    </property>
  </bean>

  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.
JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <context:property-placeholder location="jdbc.properties"/>

</beans>
```


Functionality exposed by the JdbcTemplate class

The Spring `JdbcTemplate` provides many helpful methods for the CRUD operations for the database.

Querying (select)

Here, we use the `select` command to query the database using the `JdbcTemplate` class. Depending upon the following application requirements, the database table can be queried:

- The following is a simple query to get the number of rows in a relation:

```
int rowCount = this.jdbcTemplate.queryForObject("select
count(*) from employee ", Integer.class);
```

- A simple query that uses the bind variable is shown here:

```
int countOfEmployeesNamedRavi =
this.jdbcTemplate.queryForObject(
    "select count(*) from employee where Name = ?",
    Integer.class, "Ravi");
```

- The following is a simple query for String:

```
String empName = this.jdbcTemplate.queryForObject(
    "select Name from employee where EmpId = ?",
    new Object[]{12121}, String.class);
```

- The code block to populate a domain object after querying is shown here:

```
Employee employee = this.jdbcTemplate.queryForObject(
    "select Name, Age from employee where EmpId = ?",
    new Object[]{1212},
    new RowMapper<Employee>() {
        public Employee mapRow(ResultSet rs, int
rowNum) throws SQLException {
            Employee emp = new Employee(rs.getString("Name"),
rs.getString("Age"));
            return emp;
        }
    });
```

- The code block to populate a list of the domain objects after querying is given here:

```
List<Employee> employee = this.jdbcTemplate.query(
    "select Name, Age from employee",
    new RowMapper<Employee>() {
        public Employee mapRow(ResultSet rs, int
rowNum) throws SQLException {
            Employee emp = new Employee(rs.getString("Name"),
rs.getString("Age"));
            return emp;
        }
    });
```

Apart from querying the database table, the operation for updating the record can also be performed as discussed in the next section.

Updating (Insert-Update-Delete)

When we talk about updating a record, it simply implies inserting a new record, making a change in an existing record, or deleting an existing record.

The `Update()` method is used to perform operations such as insert, update, or delete. The parameter values are usually provided as an object array or var args. Consider the following cases:

- The following shows an Insert operation:

```
this.jdbcTemplate.update("insert into employee (EmpId,
Name, Age) values (?, ?, ?)", 12121, "Ravi", "Soni");
```

- An Update operation is shown here:

```
this.jdbcTemplate.update("update employee set Name = ?
where EmpId = ?", "Shree", 12121);
```

- A Delete operation is given here:

```
this.jdbcTemplate.update("delete from employee where EmpId
= ?", Long.valueOf(empId));
```

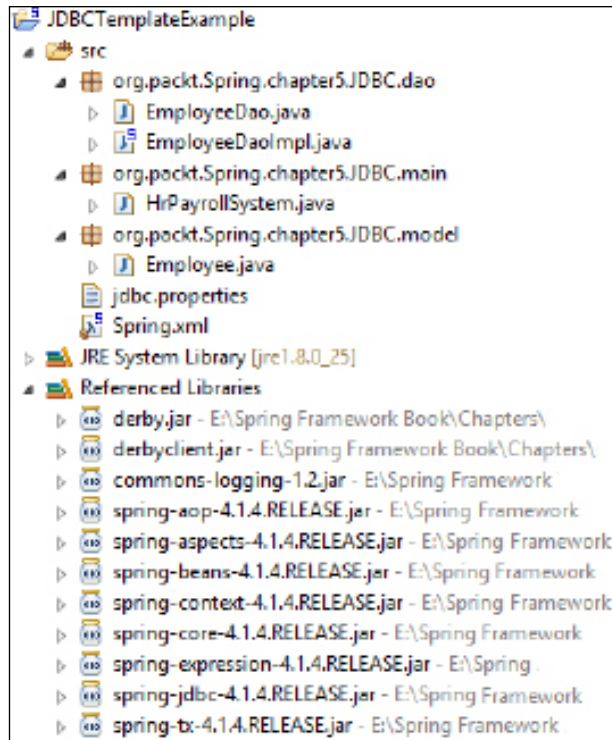
Other JdbcTemplate operations

The `execute()` method is used for executing any arbitrary SQL:

```
this.jdbcTemplate.execute("create table employee (EmpId integer,
Name varchar(30), Age integer);
```

Directory structure of the application

The final directory structure of the application is shown here



The Employee.java file

The Employee class has parameterized the constructor with three parameters, namely, empId, name, and age:

```
package org.packt.Spring.chapter5.JDBC.model;

public class Employee {

    private int empId;
    private String name;
    private int age;

    public Employee(int empId, String name, int age) {
        setEmpId(empId);
    }
}
```

```
        setName(name);  
        setAge(age);  
    }  
  
    // setter and getter
```

The EmployeeDao.java file

The EmployeeDao interface contains the declaration of a method whose implementation is provided in EmployeeDaoImpl.java:

```
package org.packt.Spring.chapter5.JDBC.dao;  
import org.packt.Spring.chapter5.JDBC.model.Employee;  
public interface EmployeeDao {  
    void createEmployee();  
    int getEmployeeCount();  
    int insertEmployee(Employee employee);  
    int deleteEmployeeById(int empId);  
    Employee getEmployeeById(int empId);  
}
```

The EmployeeDaoImpl.java file

Now let's look at the implementation of EmployeeDao, where we will use the JdbcTemplate class to execute the different types of queries:

```
package org.packt.Spring.chapter5.JDBC.dao;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.sql.Types;  
  
import org.packt.Spring.chapter5.JDBC.model.Employee;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.RowMapper;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public class EmployeeDaoImpl implements EmployeeDao {  
    @Autowired  
    private JdbcTemplate jdbcTemplate;  
  
    @Override  
    public int getEmployeeCount() {
```

```
        String sql = "select count(*) from employee";
        return jdbcTemplate.queryForInt(sql);
    }

    @Override
    public int insertEmployee(Employee employee) {
        String insertQuery = "insert into employee (EmpId, Name,
Age) values (?, ?, ?) ";
        Object[] params = new Object[] { employee.getEmpId(),
            employee.getName(), employee.getAge() };
        int[] types = new int[] { Types.INTEGER, Types.VARCHAR,
Types.INTEGER };
        return jdbcTemplate.update(insertQuery, params, types);
    }

    @Override
    public Employee getEmployeeById(int empId) {
        String query = "select * from Employee where EmpId = ?";
        // using RowMapper anonymous class, we can create a
        separate RowMapper
        // for reuse
        Employee employee = jdbcTemplate.queryForObject(query,
            new Object[] { empId }, new
        RowMapper<Employee>() {

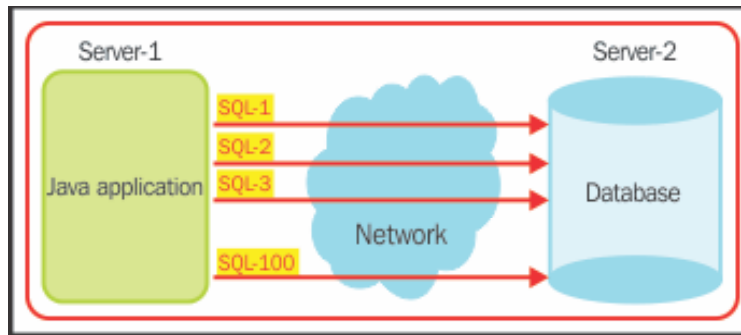
            @Override
            public Employee mapRow(ResultSet rs,
int rowNum)
                throws SQLException {
                    Employee employee = new
Employee(rs.getInt("EmpId"), rs

.getString("Name"), rs.getInt("Age"));
                    return employee;
                }
        });
        return employee;
    }

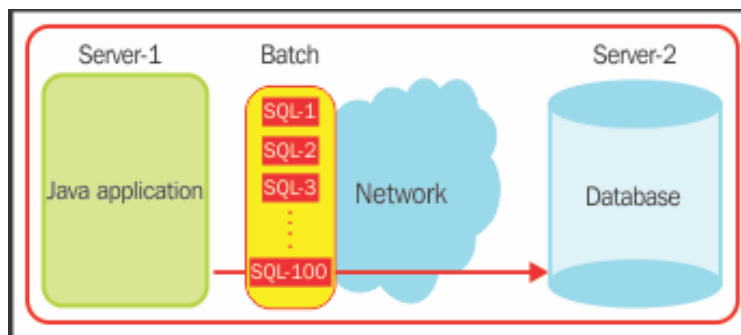
    @Override
    public int deleteEmployeeById(int empId) {
        String delQuery = "delete from employee where EmpId =
?";
        return jdbcTemplate.update(delQuery, new Object[] {
empId });
    }
}
```

JDBC batch operation in Spring

The single executable unit for performing multiple operations is known as a batch. If you batch multiple calls to the same prepared statement, then most of the JDBC drivers show improved performance. Moreover, if you group the updates into batches, then you can limit the number of round trips to the database, as shown in the following diagram:



As shown in the aforementioned figure, we have **Server-1**, where our Java application is running, and in **Server-2**, the database is running. Both the servers are situated in different locations. Let's assume that we have to execute 100 queries. Generally, we send each query from the Java application to the database server and execute them one by one. Here, we have sent the **SQL-1** query from the Java application to the database server for execution, and then the **SQL-2** query, and so on till the **SQL-100** query. So here, for 100 queries, we have to send the SQL queries from the Java application to the database server through the network. This will add a communication overhead and reduce the performance. So to improve the performance and reduce the communication overhead, we use the JDBC batch processing, as shown here:



JDBC with batch processing

In the preceding figure, we have a batch with 100 SQL queries, which will be sent from the Java application server to the database server only once, and they will still be executed. So, there is no need to send each SQL query from the Java application server to the database server. In this way, it will reduce the communication overhead and improve the performance.

The batch update operation allows you to submit multiple SQL queries to the `DataSource` for processing at once. Submitting multiple SQL queries at once instead of submitting them individually, improves the performance.

This section explains how to use an important batch update option with the `JdbcTemplate`. The `JdbcTemplate` includes a support for executing the batch of statements through a JDBC statement and through `PreparedStatement`.

The `JdbcTemplate` includes the following two overloaded `batchUpdate()` methods that support this feature:

- One method is for executing a batch of SQL statements using the JDBC statement. This method's signature is that it issues multiple SQL updates, as shown here:

```
public int[] batchUpdate(String[] sql) throws
DataAccessException
```

The following sample code shows how to use this method:

```
jdbcTemplate.batchUpdate (new String [] {
    "update emp set salary = salary * 1.5 where
empId = 10101",
    "update emp set salary = salary * 1.2 where
empId = 10231",
    "update dept set location = 'Bangalore'
where deptNo = 304"
});
```

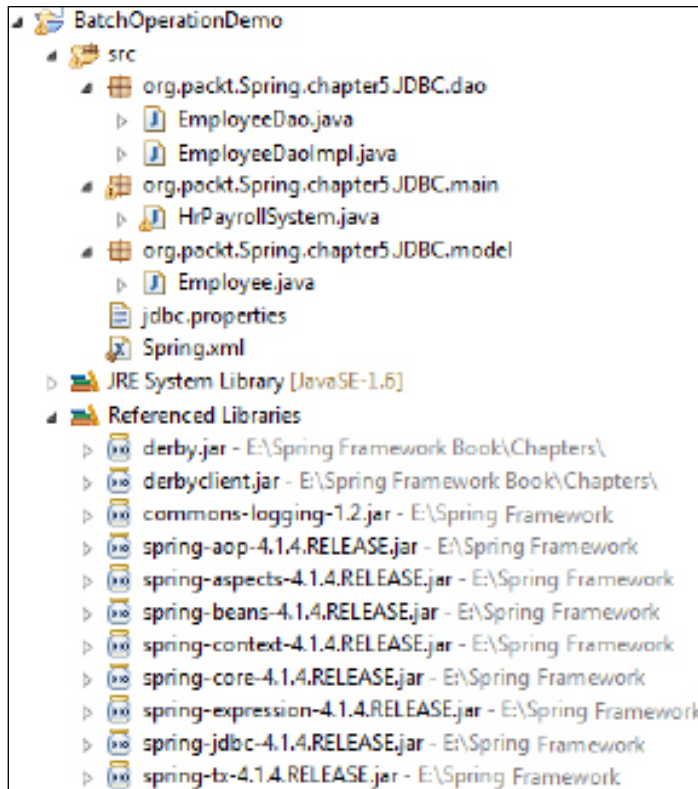
- The other method is for executing the SQL statement multiple times with different parameters using `PreparedStatement`, as shown by the following code snippet:

```
public int[] batchUpdate(String sql,
BatchPreparedStatementSetter bpss) throws
DataAccessException
```

Let's consider an example of a code, where an update batch operation performs actions.

Directory structure of the application

The final directory structure of the application is shown here:



The EmployeeDaoImpl.java file

The `EmployeeDaoImpl` class has the method `insertEmployees()` that performs the batch insert operation, as shown here:

```
package org.packt.Spring.chapter5.JDBC.dao;

import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.List;

import org.packt.Spring.chapter5.JDBC.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
```



```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class EmployeeDaoImpl {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void insertEmployees(final List<Employee> employees) {
        jdbcTemplate.batchUpdate("INSERT INTO employee "
            + "(id, name) VALUES (?, ?)",
            new BatchPreparedStatementSetter() {

                public void
setValues(PreparedStatement ps, int i)
                    throws SQLException {
                        Employee employee =
employees.get(i);
                        ps.setLong(1,
employee.getId());
                        ps.setString(2,
employee.getName());
                    }

                public int getBatchSize() {
                    return employees.size();
                }
            });
    }

    public int getEmployeeCount() {
        String sql = "select count(*) from employee";
        return jdbcTemplate.queryForInt(sql);
    }
}
```

The HrPayrollBatchUpdate.java file

The HrPayrollBatchUpdate class calls a method from EmployeeDaoImp to perform a batch update operation:

```
package org.packt.Spring.chapter5.JDBC.batchupdate;

public class HrPayrollBatchUpdate {
```

```

public static void main(String[] args) {

    ApplicationContext context = new
    ClassPathXmlApplicationContext(
        "Spring.xml");
    EmployeeDaoImp employeeDaoImp = (EmployeeDaoImp) context
        .getBean("employeeDaoImp");

    List<Employee> employeeList = new ArrayList<Employee>();
    Employee employee1 = new Employee(10001, "Ravi");
    Employee employee2 = new Employee(23330, "Kant");
    Employee employee3 = new Employee(12568, "Soni");
    employeeList.add(employee1);
    employeeList.add(employee2);
    employeeList.add(employee3);
    employeeDaoImp.insertEmployees(employeeList);
    System.out.println(employeeDaoImp.getEmployeeCount());
}
}

```

The preceding code shows how to use the `batchUpdate()` method with string and `BatchPreparedStatementSetter` for executing a SQL statement multiple times with different parameter values. In this section, we have seen how to execute batch statements using a `JdbcTemplate`.

Calling a stored procedure

A stored procedure is a group of transact SQL statements. If you have a situation where you write the same query over and over again, then you can save that specific query as a stored procedure and call it just by calling its name. Stored procedures are a block of SQL statements that are stored as basic objects within your database.

Let's take our `Employee` table that has columns as `EmpId`, `Name`, and `Age`. Let's say that we need the name and age of an employee, we will write the query as `Select Name, Age from employee`. So every time we need the name and age of the employee, we will need to write this query. Instead, we can add this query to the stored procedure and call that stored procedure rather than writing this query again and again.

The advantages and disadvantages of using the stored procedure are as follows:

Advantages	Disadvantages
Stored procedure helps in increasing the performance of an application. Stored procedures, once created, are compiled and stored in the database. And this compiled version of the stored procedures is used if an application uses the stored procedures multiple times in a single connection.	Stored procedures are difficult to debug and only a few DBMS allow you to debug it.
It helps in reducing the traffic between the application and the database server. Because, the application has to send the name and the parameter of the stored procedures rather than sending the multiple length SQL statements.	Developing and maintaining the stored procedures is not easy and leads to problems in the development and the maintenance phases, as it requires a specialized skill set, which the average developer has no interest in learning.

Using the SimpleJdbcCall class

An instance of the `SimpleJdbcCall` class is that of a multithreaded and reusable object, representing a call to a stored procedure. It provides the metadata processing to simplify the code required for accessing the basic stored procedure. While executing a call, you only have to provide the name of the stored procedure. The names of the supplied parameters are matched with the in and out parameters, specified during the declaration of a stored procedure. Here, we will discuss the calling of a stored procedure and a stored function using the `SimpleJdbcCall` class.

Calling a stored procedure

The `SimpleJdbcCall` class takes the advantage of the metadata present in the database to look up the names of the `IN` and `OUT` parameters, and thereby there is no need to explicitly declare the parameters. However, you can still declare them if you have the parameters that don't have the automatic mapping of the class, such as the array parameters.

In `MYSQL`, we declare a stored procedure named `getEmployee`, which contains an `IN` parameter `ID` and two `OUT` parameter `IDs`, named `Emp_Name` and `Emp_Age`. The query lies between `BEGIN` and `END`:

```
IN MYSQL
```

```
DROP PROCEDURE IF EXISTS getEmployee
CREATE PROCEDURE getEmployee
(
```

```

    IN id INTEGER,
    OUT Emp_Name VARCHAR(20),
    OUT Emp_Age INTEGER
)
BEGIN
    SELECT Name, Age
    INTO Emp_Name, Emp_Age
    FROM employee where EmpId = id;
END;

```

In the preceding code snippet, three parameters were specified. First was the `IN` parameter `id`, containing the ID of the employee. The remaining parameters were the `OUT` parameters, which were used for returning the data retrieved from table.

In Apache Derby, we declare a stored procedure named `getEmployee` as shown here:

```

IN Apache Derby

CREATE PROCEDURE getEmployee(IN id INTEGER, OUT name varchar(30))
LANGUAGE JAVA EXTERNAL NAME
'org.packt.Spring.chapter5.JDBC.dao.EmployeeDaoImp.getEmployee'
PARAMETER STYLE JAVA;

```

The `CREATE PROCEDURE` statement, as shown in aforementioned code snippet, allows us to create the Java stored procedures that can be called by using the `CALL PROCEDURE` statement. The `getEmployee` is a procedure name that is created in the database. The `LANGUAGE JAVA` makes the database manager call the procedure as a public static method in a Java class. The `EXTERNAL NAME 'package.class_name.method_name'` makes the `method_name` method to be called when the procedure is executed. Here, the `EXTERNAL NAME 'org.packt.Spring.chapter5.JDBC.dao.EmployeeDaoImp.getEmployee'` makes the `getEmployee` method get called during the execution of the procedure. The Java method created `org.packt.Spring.chapter5.JDBC.dao.EmployeeDaoImp.getEmployee` is specified as the `EXTERNAL NAME`.

Now, let's discuss the implementation of `SimpleJdbcCall` for calling the `getEmployee` stored procedure. The following code snippet shows us how to read the `getEmployee` stored procedure.

The EmployeeDaoImpl.java file

The following code snippet gives the `EmployeeDaoImpl.java` class:

```

package org.packt.Spring.chapter5.JDBC.dao;

import java.util.Map;

```

```
import javax.sql.DataSource;

import org.packt.Spring.chapter5.JDBC.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.stereotype.Repository;

@Repository
public class EmployeeDaoImpl implements EmployeeDao {
    @Autowired
    private DataSource dataSource;
    @Autowired
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall jdbcCall;

    public void setJdbcTemplateObject(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcCall = new SimpleJdbcCall(this.dataSource)
            .withProcedureName("getEmployee");
    }

    @Override
    public Employee getEmployee(Integer id) {
        SqlParameterSource in = new
        MapSqlParameterSource().addValue("id", id);
        Map<String, Object> simpleJdbcCallResult =
        jdbcCall.execute(in);
        Employee employee = new Employee(id,
            (String) simpleJdbcCallResult.get("name"));
        return employee;
    }
}
```

In the preceding code snippet, the instance of the `SqlParameterSource` interface was created, which contained the parameters that must match the name of the parameter declared in the stored procedure. The `execute()` method accepts the `IN` parameter as an argument and returns a map containing the `OUT` parameters, keyed by the name, as specified in the stored procedure. Here the `OUT` parameter is `name`. The retrieved value is set to the employee instance of `employee`.

Exercise

Q1. Explain the Spring JDBC packages.

Q2. What is `JdbcTemplate`?

Q3. Explain the JDBC batch operation in Spring.



The answers to these are provided in *Appendix A, Solution to Exercises*.

Summary

In this chapter, we understood the overview of database and covered the DAO design pattern. We looked at JDBC without the Spring Framework and the simplified Spring JDBC abstraction framework. We implemented the JDBC code using the Spring JDBC support. We discussed how Spring manages the `DataSource` and which data sources can be used in our applications. We also discussed the data support in the Spring application. We looked at the JDBC batch operation in Spring and calling the stored procedure by using `SimpleJdbcCall`.

In the next chapter, you will learn about ORM and understand the concept of Hibernate. Then, we will discuss the important elements of the Hibernate architecture. We will also learn how to use HQL and HCQL to query the persistent object.

4

Hibernate with Spring

While developing a real-world application using the Spring Framework, we often store and retrieve data to and from the relational database in the form of objects. These objects are non-scalar values that can't be directly stored and retrieved to and from the database, as only scalar values can be directly stored in the relational database, which is technically defined as impedance mismatch. In the previous section, we took a look at using JDBC in Spring applications.

Data persistence is the ability to preserve the state of an object so that it can regain the same state in the future. In this chapter, we will be focused on saving in-memory objects into the database using ORM tools that have wide support in Spring **Hibernate**.

As we have understood from earlier chapters, Spring uses POJO-based development and also uses declarative configuration management to overcome EJB's clumsy and heavy setup (EJB architecture was released a lot of time ago and is just not feasible).

The developer community realized that the development of data access logic could be easy using a simple, lightweight POJO-based framework. This resulted in the introduction of ORM. The objective of ORM libraries was to close the gap between the data structure in the RDBMS and the object-oriented model in Java. It helped developers focus on programming with the object model.

Hibernate is one of the most successful ORM libraries available in the open source community. It won the heart of the Java developer community with features such as its POJO-based approach, support of relationship definitions, and ease of development.

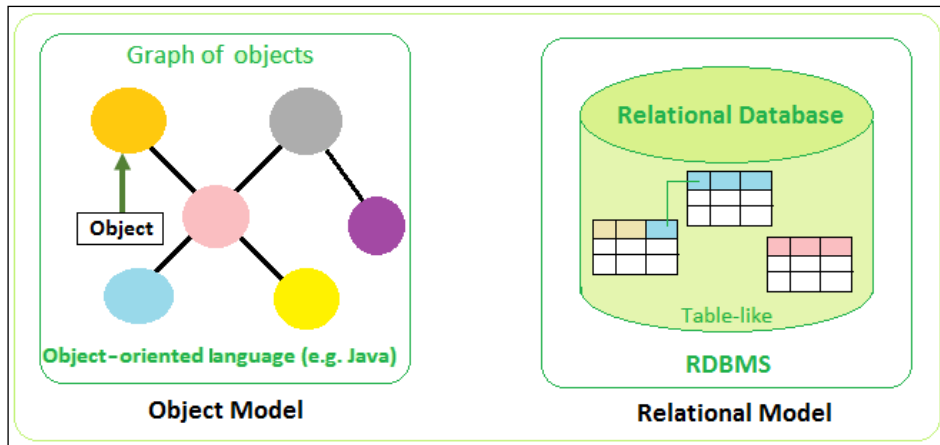
This chapter will cover the basic ideas and main use cases of Hibernate in Spring when developing data access logic. Hibernate is an extensive ORM library, so it is not possible to cover every aspect of Hibernate in just one chapter.

The list of topics that will be covered in this chapter are:

- Why Object/Relational Mapping (ORM)?
- Introducing ORM, O/RM, and O/R mapping
- Introducing Hibernate
- Integrating Hibernate with the Spring Framework
- Hibernate Criteria Query Language (HCQL)

Why Object/Relational Mapping?

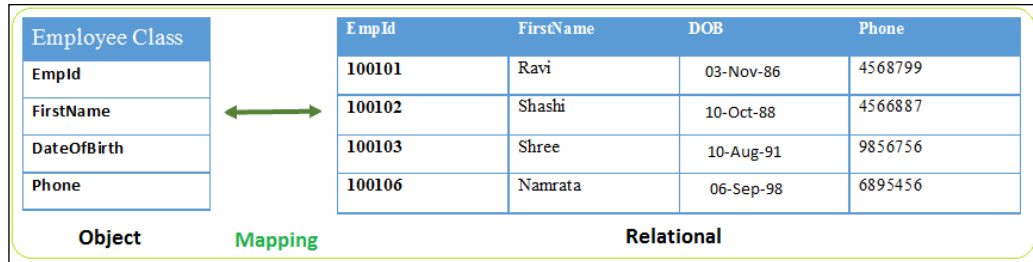
Object-oriented languages such as Java represent data as an interconnected **Graph of Objects**, whereas relational database systems represent data in a table-like format. Relational databases normally work with tables; data is stored in different types of tables. Java implements the object model whereas **relational database management systems (RDBMS)** implement the relational model. Because both the models are quite different in the way they represent data, when we load or store graphs of objects using relational databases, it causes mismatch problems. Refer to the following figure for clarity



Enterprise-level applications implemented using object-oriented languages such as Java manage the data in the form of objects. Most of these applications use relational databases such as RDBMS to maintain persistence of data in the form of tables with rows and columns. Implementing the data access layer using low-level APIs such as JDBC includes huge boilerplate code, which affects the productivity of the system, increasing the cost of application development.

Let's say we have an `Employee` class in our application, having fields named `empId`, `firstName`, `dateOfBirth`, and `phone`. In the running application, there will be many instances of this class in memory. Say we have four employee objects in memory and we want to save these employee objects into the relational database as a common database.

We will be having an `Employee` table with column names the same as the fields in the `Employee` class. Each of these employee objects contains data for a particular employee, which will be persisted as rows in that table. A class corresponds to a table and an object of this class corresponds to a row in that table, as shown here:



This is what we have followed as our traditional approach in Java applications over the years. We connect to a database using a JDBC connection and create a SQL query to perform an `INSERT` operation. So, that data will execute in the form of SQL queries to perform `INSERT`. Similarly, we create an object using setter methods after performing the `SELECT` query. By using boilerplate code, the object will get converted into a data model and vice versa, which results in a painful mapping process. This is a common problem in every Java application that has a persistence layer and that connects to the database in order to save and retrieve values.

Mapping relationships is another issue that needs to be addressed. Let's say we have another table called `Address` table and an object called `address` object. Let's also say the `employee` object has a reference to this `address` object. In this case, the primary key of the `Address` table will be mapped to a foreign key of the `Employee` table.

Another issue that needs to be addressed is data types. Let's say we have an object with a Boolean data type whereas most of the database doesn't have Boolean data type and probably used as char for Y/N or integer for 0/1, which need to be handled during data type conversion while writing code.

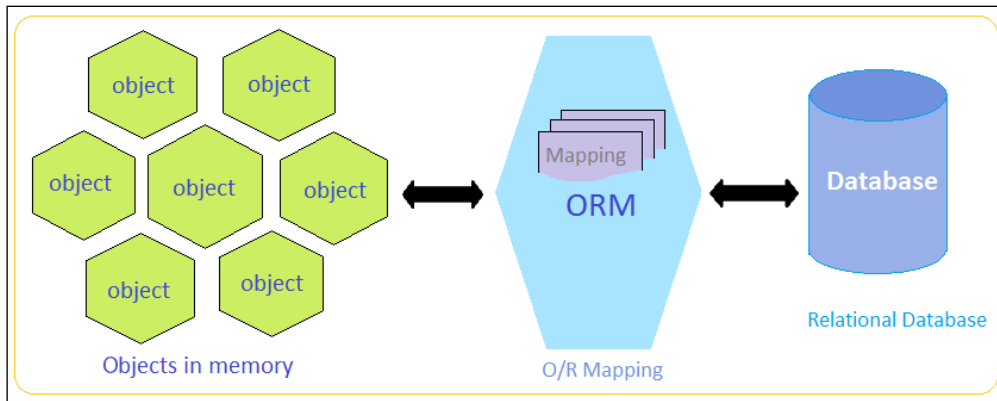
Managing changes to object state is another issue that needs to be addressed. If there are some changes to object state, then we need to manually execute the procedure to make these changes and we also need to reframe the SQL queries and update the database by ourselves.

To solve these problems, we need a customizable generic system that can take the responsibility of filling the gap between the object and relational models for our application. This requirement has resulted in the introduction of ORM, which provides an elegant way to handle the mentioned issues.

Introducing ORM, O/RM, and O/R mapping

ORM is the process of persisting objects in a relational database such as RDBMS. ORM bridges the gap between object and relational schemas, allowing object-oriented applications to persist objects directly without having the need to convert objects to and from a relational format.

ORM creates a virtual object database that can be accessed via a programming language and simplifies the data access layer of complex enterprise applications using a relational database as its persistence store. ORM simplifies the job of implementing the data access layer for enterprise applications implemented using object-oriented programming languages and the relational database as its persistence store, as illustrated by the following figure



ORM is about mapping object representations to JDBC statement parameters and in turn mapping JDBC query results back to object representations. Database columns are mapped to the instance fields of domain objects or JavaBeans' properties

Usually, ORM doesn't work at the SQL level but rather refers its own Object Query Language, which gets translated into SQL at runtime. The mapping information is kept as metadata (in XML files or as annotations on mapped objects), which defines how to map a persistent class and its fields into database tables and their columns. The database dialect is configured to address database specifics. For example ID generation is configured in the metadata and is automatically translated into sequences or autoincrement columns.

Until now, we have understood how ORM implementations in Java help us to quickly implement a reliable data access layer to concentrate on other tiers of the application. In the next section, we will understand the features of Hibernate and their uses in a Spring application.

Introducing Hibernate

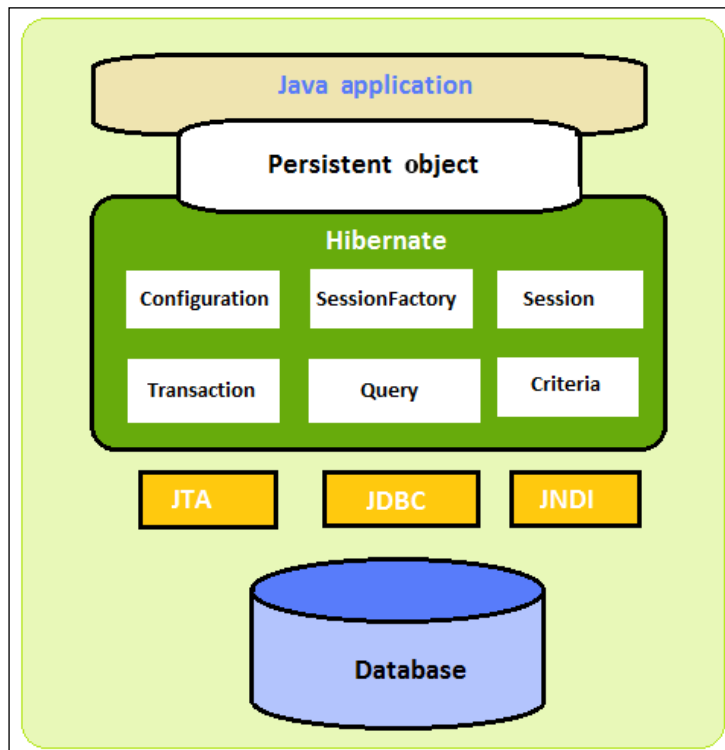
Hibernate, by definition, is an ORM solution for Java. Hibernate is an open source, full-fledged persistence framework. It is used to map **plain old Java objects (POJOs)** to the tables of a relational database and vice versa. Hibernate is used to persist application data into a data layer. Hibernate implements **Java Persistence API (JPA)**, which is a set of standards that has been prescribed for any persistence implementation and that needs to be met in order to get certified as a Java persistent API implementation.

Hibernate sits between Java objects in memory and the relational database server to handle the persistence of objects based on O/R mapping. Hibernate supports almost all relational database engines such as the HSQL database engine, MySQL, PostgreSQL, Oracle, and so on.

The object query language used by Hibernate is called **Hibernate Query Language (HQL)**. HQL is a SQL-like textual query language that works at the class- or field level. Let's start learning about the architecture of Hibernate.

Hibernate architecture

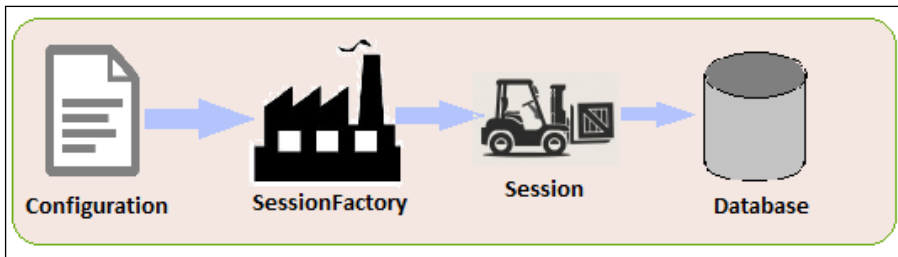
In this section, we will discuss all the important elements of the Hibernate system and see how they fit into its architecture. The following figure shows the Hibernate architecture:



Hibernate makes use of various existing Java APIs such as **Java Database Connectivity (JDBC)**, **Java Naming and Directory Interface (JNDI)**, and **Java Transaction API (JTA)**. JDBC supports functionality common to relational databases, which allows almost any database with a JDBC driver to be supported by Hibernate, whereas JTA and JNDI allow Hibernate to be integrated with Java EE application servers. The basic elements of the Hibernate architecture are described in the following sections.

Configuration

The `org.hibernate.cfg.Configuration` class is the basic element of the Hibernate API, which allows us to build `SessionFactory`. Configuration can be thought of as the factory class that can produce `SessionFactory`. The first object of Hibernate is the configuration object, created only once during the initialization of the application. The configuration object encapsulates the Hibernate configuration details such as connection properties and dialect, which are used to build `SessionFactory` as shown in the following figure



The `hibernate.properties` and `hibernate.cfg.xml` files are configurations files that are supported by Hibernate. We can use the `hibernate.properties` file to specify the default values for the new configuration object

SessionFactory

The `org.hibernate.SessionFactory` interface provides an abstraction for the application to obtain the Hibernate session object. The `SessionFactory` initialization process includes various operations that consume huge resources and extra time, so it is generally recommended to use a single `SessionFactory` per JVM instance. For each database, we need to have one `SessionFactory` using a separate configuration file. So we have to create multiple `SessionFactory` if we are using multiple databases.

The `SessionFactory` is a heavyweight and immutable towards the application; that is, it is a thread safe object. It is mostly configured as a singleton in an application so that there will be only one object per application. It is usually created during the startup of an application and is kept for later reference. The `SessionFactory` is used by all threads of the application. We can open multiple sessions using a single `SessionFactory`.

Session

The `org.hibernate.Session` interface is an interface between the Hibernate system and the application. It is used to get the connection with a database. It is light weight and is initiated each time an interaction is needed with the database.

Session objects are not usually thread safe and it is recommended to obtain a separate session for each thread or transaction. After we are done using session, it has to be closed to release all the resources such as cached entity objects and the JDBC connection.

The `Session` interface provides an abstraction for Java application to perform CRUD operations on the instance of mapped persistent classes. We will look into the methods provided by the `Session` interface in a later section of this chapter.

Transaction

The `Transaction` interface is an optional interface that represents a unit of work with the database. It is supported by most RDBMS systems. In Hibernate, `Transaction` is handled by the underlying transaction manager.

Query

The `org.hibernate.Query` interface provides an abstraction to execute the Hibernate query and to retrieve the results. The `Query` object represents the Hibernate query built using the HQL. We will learn about the `Query` interface in more detail in a later section of this chapter.

Criteria

The `org.hibernate.Criteria` interface is an interface to use the criterion API and is used to create and execute object-oriented criteria queries, which is an alternative to HQL or SQL.

The Persistent object

Persistent classes are the entity classes in an application. Persistent objects are objects that are managed to be in the persistent state. Persistent objects are associated with exactly one `org.hibernate.Session`. And once the `org.hibernate.Session` is closed, these objects will be detached and will be free to be used in any layer of the application.

Integrating Hibernate with the Spring Framework

While using the Hibernate framework, you do not write the code to manage the connection or to deal with statements and result sets. Instead, all the details for accessing a particular data source are configured in the XML files and/or in the Java annotations.

While integrating the Hibernate framework with the Spring Framework, the business objects are configured with the help of the IoC container and can be externalized from the application code. Hibernate objects can be used as Spring beans in your application and you can avail all the benefits of the Spring Framework.

In this section, we will set up the Hibernate environment and create a Spring Hibernate project in STS. The simplest way to integrate Hibernate with Spring is to have a bean for `SessionFactory` and make it a singleton and the DAOs classes just get that bean and inject its dependency and get the session from the `SessionFactory`. The first step in creating a Spring Hibernate project is to integrate Hibernate and connect with the database.

Sample data model for example code

In this chapter, we will use a PostgreSQL database. Please refer to <http://www.postgresqltutorial.com/install-postgresql/> to set up a PostgreSQL database server on your machine and download the JDBC driver for the PostgreSQL database; we have used the `postgresql-9.3-1102.jdbc3.jar` JDBC connector for PostgreSQL.

We will create a database named `ehrpayroll_db` that will contain a table named `employee` and will populate dummy data to the table. The following is a sample data creation script for a PostgreSQL database.

Let's first create a database for our project in the PostgreSQL database

1. Type in the following script to create a database named `ehrpayroll_db`:

```
CREATE DATABASE ehrpayroll_db
```

2. Now enter the given script to create a table named `EMPLOYEE_INFO`:

```
CREATE TABLE EMPLOYEE_INFO (
    ID serial NOT NULL Primary key,
    FIRST_NAME varchar(30) not null,
    LAST_NAME varchar(30) not null,
```



```
JOB_TITLE varchar(100) not null,  
DEPARTMENT varchar(100) not null,  
SALARY INTEGER  
);
```

3. The next script helps you populate the data for the table employee:

```
INSERT INTO EMPLOYEE_INFO  
(FIRST_NAME, LAST_NAME, JOB_TITLE, DEPARTMENT, SALARY)  
VALUES  
('RAVI', 'SONI', 'AUTHOR', 'TECHNOLOGY', 5000);
```

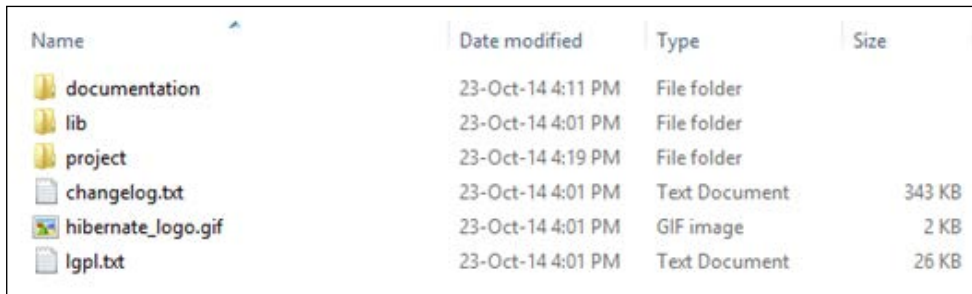
The following figure shows the created table with the data inserted:

	id integer	first_name character varying(30)	last_name character varying(30)	job_title character varying(100)	department character varying(100)	salary integer
1	1	RAVI	SONI	AUTHOR	TECHNOLOGY	5000

Integrating Hibernate







To integrate Hibernate, we need to perform these steps:

1. Download the Hibernate JAR and include them into the classpath. Download (.zip file for Windows) the latest version of Hibernate from <http://www.hibernate.org/downloads>. Once you unzip the downloaded ZIP file, the directory structure will appear as shown in the following screenshot:



Name	Date modified	Type	Size
documentation	23-Oct-14 4:11 PM	File folder	
lib	23-Oct-14 4:01 PM	File folder	
project	23-Oct-14 4:19 PM	File folder	
changelog.txt	23-Oct-14 4:01 PM	Text Document	343 KB
hibernate_logo.gif	23-Oct-14 4:01 PM	GIF image	2 KB
lgpl.txt	23-Oct-14 4:01 PM	Text Document	26 KB

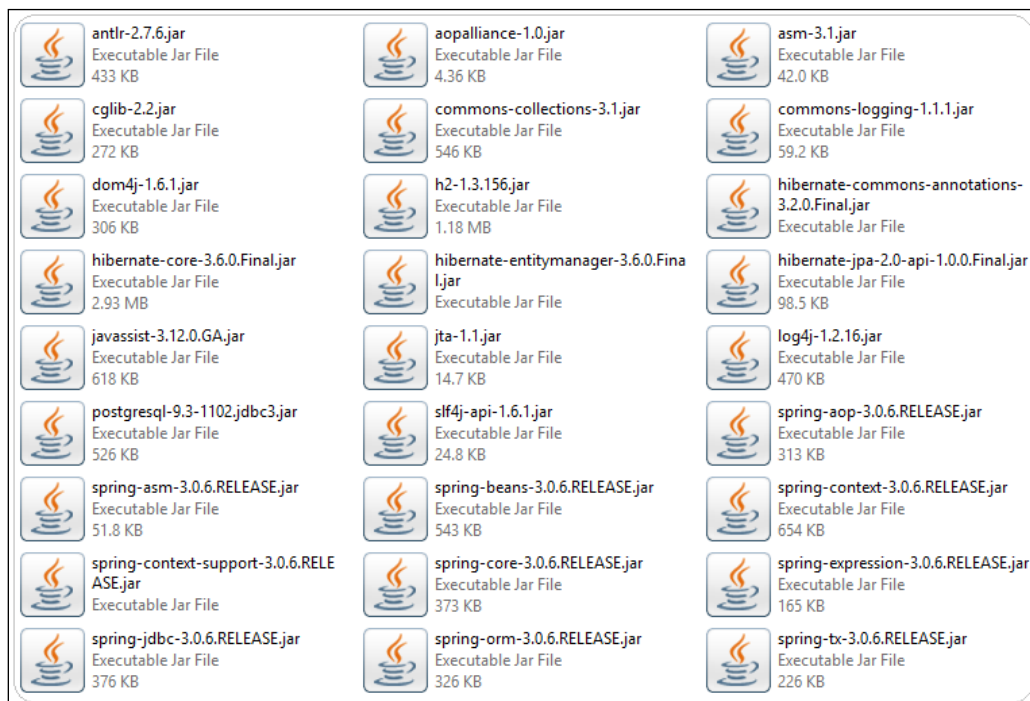
2. Inside the lib directory, there will be a lot of directories that contain Hibernate-related JARs, as shown in the following screenshot. The required folder contains all the JARs you need to create a basic Java application.

Name	Date modified	Type	Size
 envers	23-Oct-14 4:01 PM	File folder	
 jpa	23-Oct-14 4:01 PM	File folder	
 jpa-metamodel-generator	23-Oct-14 4:01 PM	File folder	
 optional	23-Oct-14 4:01 PM	File folder	
 osgi	23-Oct-14 4:01 PM	File folder	
 required	23-Oct-14 4:01 PM	File folder	

Once you have downloaded the Hibernate libraries, you can create a new Spring project and add Hibernate libraries to this project using Java Build Path.

Required JARs for the Spring-Hibernate project

We need to add the JARs required to create our Spring-Hibernate projects. These are shown in the following screenshot:



Configuring Hibernate SessionFactory in Spring

The Spring Framework lets us define resources such as JDBC `DataSource` or Hibernate `SessionFactory` as a Spring bean in an application context, which prevents the need for hardcoded resource lookups for application objects. This defined Spring bean references are used by application objects that need to access resources to receive the predefined instances

The `Session` interface in the Hibernate API provides methods to find, save, and delete objects in a relational database. The Hibernate session is created by first creating the `SessionFactory`. The Spring Framework provides a number of classes to configure Hibernate `SessionFactory` as a Spring bean containing the desired properties.

For session creation, the Spring API provides the implementation of the `AbstractSessionFactoryBean` subclass: the `LocalSessionFactoryBean` class and the `AnnotationSessionFactoryBean` class. Since we will be using annotation style, we will use the `AnnotationSessionFactoryBean` class, which supports annotation metadata for mappings. `AnnotationSessionFactoryBean` extends the `LocalSessionFactoryBean` class, so it has all the basic properties of Hibernate integration.

In the configuration file, we have to declare the `sessionFactory` bean and set `dataSource`, `packagesToScan` or `annotatedClasses`, and `hibernateProperties`. Let's take a look at these in detail:

- The `dataSource` property sets the name of the data source to be accessed by the underlying application.
- The `packagesToScan` property instructs Hibernate to scan the domain object with the ORM annotation under the specified package. The `annotatedClasses` property instructs Hibernate to for the ORM-annotated class.
- The `hibernateProperties` property sets the configuration details for Hibernate. We have defined only a few important properties out of many configuration parameters that should be provided for every application

The following table describes these properties:

Property	Description
<code>hibernate.dialect</code>	Hibernate uses this property to generate the appropriate SQL optimized for the chosen relational database. Hibernate supports SQL dialects for many databases, and the major dialects include <code>PostgreSQLDialect</code> , <code>MySQLDialect</code> , <code>H2Dialect</code> , <code>Oracle10gDialect</code> , and so on.
<code>hibernate.max_fetch_depth</code>	This property is used to set the maximum depth for the outer join when the mapping object is associated with other mapped objects. This property is used to determine the number of associations Hibernate will traverse by join when fetching data. The recommended value lies between 0 and 3.
<code>hibernate.jdbc.fetch_size</code>	This property is used to set the total number of rows that can be retrieved by each JDBC fetch.
<code>hibernate.show_sql</code>	This property file is used to output all SQL to the log file or console, which is an alternative to set log to debug and troubleshooting process. It can be set to either <code>True</code> or <code>False</code> .

Refer to the Hibernate reference manual for the full list of Hibernate properties at <http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/session-configuration.html>.

XML Spring configuration for Hibernate

Spring beans, the data source, a `SessionFactory`, and a transaction manager bean are configured in the `app-context.xml` file. You should adapt your Hibernate beans according to the project requirements.

Here is an implementation of `app-context.xml`. In the following configuration file we have declared several beans to support the `Hibernate SessionFactory`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="
```

```
http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
3.0.xsd">
```

In the following code snippet, we have instructed Spring to scan the component under the package `org.packt.spring.chapter6.hibernate` using `component-scan`:

```
<context:annotation-config />
<context:component-scan base-
package="org.packt.spring.chapter6.hibernate" />
```

The `property-placeholder` will refer to the `hibernate.properties` file, as shown in the following code snippet:

```
<context:property-placeholder
location="classpath:/META-
INF/spring/hibernate.properties" />
```

In the following code snippet, the `dataSource` bean is declared to provide database connection details to Hibernate:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
    <property name="driverClassName"
value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
```

The `sessionFactory` bean is declared in the following code snippet.

The Hibernate `sessionFactory` bean is the most important part. We have used `AnnotationSessionFactoryBean` to support the Hibernate annotation. We have injected the `dataSource` bean into `sessionFactory`. We have instructed Hibernate to scan for the ORM annotated object. And then we have provided the configuration details for Hibernate using `hibernateProperties`, as shown here:

```

<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.
annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses"
value="org.packt.spring.chapter6.hibernate.model.Employee" />
    <property name="hibernateProperties">
        <props>
            <prop
key="hibernate.dialect">${hibernate.dialect}</prop>
            <prop
key="hibernate.show_sql">${hibernate.show_sql}</prop>
        </props>
    </property>
</bean>

```

In the following code snippet, we have declared the `transactionManager` bean. To access transactional data, `SessionFactory` requires a transaction manager. The transaction manager provided by Spring specifically for Hibernate 3 is `org.springframework.orm.hibernate3.HibernateTransactionManager`:

```

<bean id="transactionManager"
class="org.springframework.orm.hibernate3.
HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

```

The `<tx:annotation-driven>` is declared in the following code snippet to support transaction demarcation requirements using annotations:

```

<tx:annotation-driven transaction-manager=
"transactionManager" />
</beans>

```

hibernate.properties

The Hibernate- and JDBC-specific properties are stored in a `hibernate.properties` file, as follows

```

# JDBC Properties
jdbc.driverClassName=org.postgresql.Driver
jdbc.url=jdbc:postgresql://localhost:5432/ehrpayroll_db
jdbc.username=postgres

```

```
jdbc.password=sa

# Hibernate Properties
hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
hibernate.show_sql=true
```

Annotated domain model class

The Java persistent model establishes the static relationships of the persistence model by defining the `Entity` component. The API defines the entity class as the object tier of a table in the database tier. An entity instance is defined as the object tier equivalent of a row in a database table.

The following is a table that maps Object Tier elements to Database Tier elements:

Object Tier element	Database Tier element
Entity class	Database table
Field of entity class	Database table column
Entity instance	Database table row

Hibernate annotation provides the metadata for object and relational table mapping. This metadata is clubbed into a POJO file that helps users understand the code inside POJO as well as the table structure simultaneously while developing. Hibernate provides JPA implementation, which allows the user to use JPA annotation in model beans. The JPA annotations are explained in the following table:

JPA annotation	Description
@Entity	The <code>javax.persistence.Entity</code> annotation declares a class as an entity bean that can be persisted by Hibernate, since Hibernate provides JPA implementation.
@Table	The <code>javax.persistence.Table</code> annotation can be used to define table mapping. It provides four attributes that allows us to override the table name, its catalogue, and its schema.
@Id	The <code>javax.persistence.Id</code> annotation is used to define the primary key, and it will automatically determine the appropriate primary key generation strategy to be used.

JPA annotation	Description
@GeneratedValue	<ul style="list-style-type: none"> The <code>javax.persistence.GeneratedValue</code> annotation is used to define that the field will be autogenerated It takes two parameters, strategy and generator The <code> GenerationType.IDENTITY</code> strategy is used so that the generated id value is mapped to the bean and can be retrieved by the Java program
@Column	<ul style="list-style-type: none"> The <code>javax.persistence.Column</code> annotation is used to map the field with the table column We can also specify length, nullable, and uniqueness for the bean properties

Now it's time to write `Employee.java`. This class will be mapped to the `Employee` table in the database using Hibernate. The `Employee` class fields are annotated with JPA annotations so that we don't need to provide mapping in a separate XML file. It should be noted that Hibernate puts emphasis on overriding the `equals()` and `hashCode()` methods of a persistent class when used in collections (such as a list or set) because internally Hibernate works with the objects in the session and cache. It is recommended to implement the `equals()` and `hashCode()` methods using a real-world key, which is a key that would identify the instance in the real world, as shown here:

```
package org.packt.spring.chapter6.hibernate.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "EMPLOYEE_INFO")
public class Employee {

    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
```



```
@Column(name = "FIRST_NAME")
private String firstName;

@Column(name = "LAST_NAME")
private String lastName;

@Column(name = "JOB_TITLE")
private String jobTitle;

@Column(name = "DEPARTMENT")
private String department;

@Column(name = "SALARY")
private int salary;

// constructor and setter and getter

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Employee)) {
        return false;
    }
    Employee employee = (Employee) obj;
    if (firstName != null ?
!firstName.equals(employee.firstName)
        : employee.firstName != null) {
        return false;
    } else {
        return true;
    }
}

@Override
public int hashCode() {
    return firstName != null ? firstName.hashCode() : 0;
}

@Override
public String toString() {
    return "Employee [id=" + id + ", name=" + firstName + " "
+ lastName
```

```

+ department
    + ", jobTitle=" + jobTitle + " department="
    + " salary=" + salary + "];";
}
}

```

In the preceding code snippet:

- The `Employee` class is annotated with the `@Entity` annotation, which will define this class as a mapped entity class. The `Employee` class is also annotated with the `@Table` annotation that defines the table name in the database with which this entity class will map.
- The ID is annotated with the `@ID` annotation, which represents that ID is the primary key of the object. Hibernate will generate the ID value based on the `@GeneratedValue` annotation. The `GenerationType.IDENTITY` strategy reflects that the ID will be generated by the backend (the ID column of the `EMPLOYEE_INFO` table is the primary key with `SERIAL` specified, which means the value of ID will be generated and assigned by the database during the insert operation) during insert.
- The name and e-mail are annotated with the `@Column` annotation.
- If the type and attribute names are exactly the same as the name of table and column, then we can skip the name of the table and column from the annotation.

The Hibernate sessions

The `Session` interface is an important interface that is required while interacting with a database in Hibernate. The `Session` interface is obtained from `SessionFactory`. The `Session` object is light weight and can be used to attain a physical connection with a database. It is initiated each time an interaction needs to happen with a database. Also, persistent objects are saved and retrieved through a `Session` object. It is not usually thread safe, so avoid keeping it open for a long time. They should be created and destroyed as needed. The `Session` interface offers create, delete, and read operations for instances of mapped entity classes.

Instances may exist in one of the following three states at a given point in time:

- **Transient:** This state represents a new instance of persistence class that has no representation in a database and is not associated with `Session`.

- **Persistent:** This state represents that the instance of a persistence class has a representation in the database.
- **Detached:** In this state, the persistent object will be detached from the database. This state will be reached once the `Hibernate Session` will be closed.

The Session interface methods

The `Session` interface provides a numbers of method such as `beginTransaction()`, `createCriteria()`, `save()`, `delete()`, and so on, which you can read about at http://www.tutorialspoint.com/hibernate/hibernate_sessions.htm.

Persistence layer – implement DAOs

The persistence layer will have the DAO. Let's create DAO classes that will interact with the database using the `Hibernate SessionFactory`. The `SessionFactory` implementation will be injected into the reference variable at runtime using Spring's **Inversion of Control (IoC)** feature.

The EmployeeDao interface

The `EmployeeDao` interface declares two methods named `getAllEmployees()` and `insertEmployee()`, as shown here:

```
package org.packt.spring.chapter6.hibernate.dao;

import java.util.List;
import org.packt.spring.chapter6.hibernate.model.Employee;

public interface EmployeeDao {

    // to get all employees
    public List<Employee> getAllEmployees();

    // to insert new employee
    public void insertEmployee(Employee employee);
}
```

The EmployeeDaoImpl class

The `EmployeeDaoImpl` class is annotated with `@Repository`, which indicates that this class is a DAO. It also has the `@Transactional(readOnly = true)` annotation, which configures this class and all its methods for read-only access:

```

package org.packt.spring.chapter6.hibernate.dao;

import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.packt.spring.chapter6.hibernate.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
@Transactional(readOnly = true)
public class EmployeeDaoImpl implements EmployeeDao {

    @Autowired
    private SessionFactory sessionFactory;

    @SuppressWarnings("unchecked")
    public List<Employee> getAllEmployees() {
        Session session = sessionFactory.openSession();
        String hql = "FROM Employee";
        Query query = session.createQuery(hql);
        List<Employee> emList = query.list();
        return emList;
    }

    @Transactional(readOnly = false)
    public void insertEmployee(Employee employee) {
        Session session = sessionFactory.openSession();
        session.save(employee);
    }
}

```

To get a `SessionFactory`, we declare a member variable named `sessionFactory` of type `SessionFactory` and annotated with the `@Autowired` annotation that automatically initializes the `SessionFactory`. The next step is to get the session from the `sessionFactory`.

In order to use Hibernate in the `getAllEmployees()` and `insertEmployees()` method, we use a session object. The session object is obtained from a `SessionFactory`. Using this session object, we can use the `createQuery` method to create queries and run them.

When we are finished with the session, we should close it. We needn't close it by ourselves; the Spring Framework will do this for us.

Let's understand the methods defined in `EmployeeDaoImpl` class in more detail:

- Querying the database – `getAllEmployees()`: The method `getAllEmployees()` will fetch all the employee details from the `Employee` table in the database and return the list of employees.

This method will get `Hibernate Session` (`Session` is the main runtime interface between Java application and Hibernate) from `SessionFactory` using the `openSession()` method of the `SessionFactory` class. This session will use the query object to call the `list()` method that will fetch employees.
- Inserting new record – `insertEmployee()`: The method `insertEmployee()` will insert a new record to the `Employee` table. This method will use the `save()` method defined in the `Hibernate Session` to perform the `INSERT` operation.

Annotate this method with `@Transactional(readOnly = false)`, which will allow us to perform the `INSERT` operation.

Service layer – implement services

We have defined the Service layer, which seems redundant in this demo due to the lack of complexity. This layer will simply take a call from the controller (in Spring MVC) or from the main method and pass this call to the DAOs layer.

The EmployeeService interface

The `EmployeeService` interface declares two methods named `getAllEmployees()` and `insertEmployee()`, as shown here:

```
package org.packt.spring.chapter6.hibernate.service;

import java.util.List;
import org.packt.spring.chapter6.hibernate.model.Employee;

public interface EmployeeService {
    public List<Employee> getAllEmployees();
    public void insertEmployee(Employee employee);
}
```

The EmployeeServiceImpl class

The `EmployeeServiceImpl` class will implement the `EmployeeService` interface and provide a definition for the methods declared in the interface. This class has declared a member variable named `EmployeeDAO` and annotated it with the `@Autowired` annotation. This class is annotated with the `@Service` annotation, which makes this class a service class:

```
package org.packt.spring.chapter6.hibernate.service;

import java.util.List;

import org.packt.spring.chapter6.hibernate.dao.EmployeeDao;
import org.packt.spring.chapter6.hibernate.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EmployeeServiceImpl implements EmployeeService {

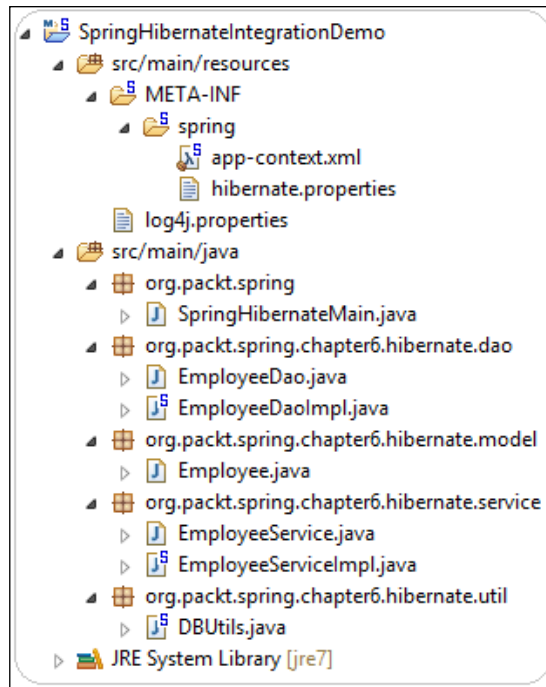
    @Autowired
    private EmployeeDao employeeDao;

    public List<Employee> getAllEmployees() {
        List<Employee> emList = employeeDao.getAllEmployees();
        return emList;
    }

    public void insertEmployee(Employee employee) {
        employeeDao.insertEmployee(employee);
    }
}
```

Directory structure of the application

The final directory structure of the application is as follows:



Running the application

Once we are done with the preceding configuration, we can write a main method to store values from the `Employee` object to the database.

The DBUtils class

We have created a `DBUtils` class annotated with the `@Component` annotation to register this class to the Spring container as a bean. This class defined a method named `initialize()` and annotated it with the `@PostConstruct` annotation.

The `@PostConstruct` annotation does not belong to Spring, it's located in the J2EE library: `common-annotations.jar`. The `@PostConstruct` annotation is a shared annotation that is part of a JSR for basic annotations. It comes with Java SE 6 or newer versions. The `commons-annotations.jar` is the final product of the JSR API. The `@PostConstruct` annotation defines a method that will be called after a bean has been fully initialized. In other words, it will be called after bean construction and the injection of all dependencies.

The `initialize()` method will get the database connection and create a table `EMPLOYEE` and insert dummy data to this table. This class has been used in this project to prevent exceptions in case we miss out on creating a table in the database. In a real-world application, we don't need this class:

```
package org.packt.spring.chapter6.hibernate.util;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

import javax.annotation.PostConstruct;
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class DBUtils {

    @Autowired
    private DataSource dataSource;

    @PostConstruct
    public void initialize() {
        try {
            Connection connection =
dataSource.getConnection();
            Statement statement =
connection.createStatement();

            statement.execute("DROP TABLE IF EXISTS
EMPLOYEE_INFO");

            statement.executeUpdate("CREATE TABLE
EMPLOYEE_INFO(" +
                "ID serial NOT NULL Primary key, " +
                "FIRST_NAME varchar(30) not null, " +
                "LAST_NAME varchar(30) not null, " +
                "JOB_TITLE varchar(100) not null, " +
                "DEPARTMENT varchar(100) not null, " +
                "SALARY INTEGER)");

            statement.executeUpdate("INSERT INTO EMPLOYEE_INFO
"
```



```
        + "(FIRST_NAME, LAST_NAME, JOB_TITLE,  
DEPARTMENT, SALARY) "  
        + "VALUES " + "('RAVI', 'SONI', 'AUTHOR',  
'TECHNOLOGY', 5000)";  
  
        statement.close();  
        connection.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}  
}
```

The SpringHibernateMain class

The SpringHibernateMain class contains the main method. The ApplicationContext will initialize the container with the app-context.xml file we define

```
package org.packt.spring;  
  
import org.packt.spring.chapter6.hibernate.model.Employee;  
import org.packt.spring.chapter6.hibernate.service.EmployeeService;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplication  
Context;  
  
public class SpringHibernateMain {  
  
    public static void main(String[] args) {  
  
        ApplicationContext context = new  
ClassPathXmlApplicationContext(  
            "/META-INF/spring/app-context.xml");  
  
        EmployeeService employeeService = context.getBean(  
            "employeeServiceImpl",  
EmployeeService.class);  
  
        // insert employee  
        Employee emp = new Employee();  
        emp.setFirstName("Shree");  
        emp.setLastName("Kant");  
        emp.setJobTitle("Software Engineer");  
        emp.setDepartment("Technology");  
    }  
}
```

```

emp.setSalary(3000);
employeeService.insertEmployee(emp);

// fetch all employee
for (Employee employee :
employeeService.getAllEmployees())
    System.out.println(employee);
}
}

```

Output to console

Once you run the application, the following output will be expected:

```

Hibernate: insert into EMPLOYEE_INFO (DEPARTMENT, FIRST_NAME,
JOB_TITLE, LAST_NAME, SALARY) values (?, ?, ?, ?, ?)

```

```

Hibernate: select employee0_.ID as ID0_, employee0_.DEPARTMENT as
DEPARTMENT0_, employee0_.FIRST_NAME as FIRST3_0_, employee0_.JOB_TITLE
as JOB4_0_, employee0_.LAST_NAME as LAST5_0_,
employee0_.SALARY as SALARY0_ from EMPLOYEE_INFO employee0_

```

```

Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]

```

```

Employee [id=2, name=Shree Kant, jobTitle=Software Engineer
department=Technology salary=3000]

```

Populated data in the Employee table

Once the application has been run successfully, the updated Employee table with all the data will be as shown here:

	id integer	first_name character varying(30)	last_name character varying(30)	job_title character varying(100)	department character varying(100)	salary integer
1	1	RAVI	SONI	AUTHOR	TECHNOLOGY	5000
2	2	Shree	Kant	Software Engineer	Technology	3000

In the previous sections, we discussed mapping persistent objects using Hibernate. In the next section, we will understand HQL. Hibernate is engineered around the object model and provides a powerful query language named HQL to define our queries so we don't need to construct SQL to interact with the database. HQL is similar to SQL except that we will use objects instead of table names.

Hibernate Query Language

Hibernate Query Language is an object-oriented query language that works on persistence objects and their properties instead of operating on tables and columns. Hibernate will translate HQL queries into conventional SQL queries during the interaction with a database.

Even though you can use SQL queries using native SQL directly with Hibernate, it is recommended that you use HQL to get the benefits of Hibernate's SQL generation and caching strategies.

In HQL, keywords such as `SELECT`, `FROM`, `WHERE`, `GROUP BY`, and so on are not case sensitive but properties such as table and column names are case sensitive. So `org.packt.spring.chapter6.hibernate.model.Employee` is not same as `org.packt.spring.chapter6.hibernate.model.EMPLOYEE`, whereas `SELECT` is similar to `Select`.

The Query interface

To use HQL, we need to use Query object. The Query interface is an object-oriented representation of HQL. The Query object can be obtained by calling the `createQuery()` method of the Session interface. The Query interface provides a number of methods such as `executeUpdate()`, `list()`, `setFirstResult()`, `setMaxResult()`, and so on. The following code snippet uses HQL to get all records:

```
@Transactional
public List<Employee> getAllEmployees() {
    Session session = sessionFactory.openSession();

    String hql = "FROM Employee";
    Query query = session.createQuery(hql);

    <Employee> emList = query.list();
    return emList;
}
```

Database operation using HQL

HQL supports clauses to perform database operation. Let's have a look at a few clauses.

The FROM clause

The `FROM` clause is used to load complete persistence objects into memory. The `FROM` clause is the same as the `SELECT` clause in SQL, as shown in the following table:

HQL	SQL
<code>FROM Employee</code>	<code>SELECT * from Employee</code>

The syntax to use the `FROM` clause is as follows:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

We can specify the package and class name if needed to fully qualify the class name, as follows:

```
String hql = "FROM org.packt.spring.chapter6.hibernate.model.
Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

The expected output to the console is:

```
Hibernate: select employee0_.ID as ID0_, employee0_.DEPARTMENT as
DEPARTMENT0_, employee0_.FIRST_NAME as FIRST3_0_,
employee0_.JOB_TITLE as JOB4_0_, employee0_.LAST_NAME as LAST5_0_,
employee0_.SALARY as SALARY0_ from EMPLOYEE_INFO employee0_
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer
department=Technology salary=3000]
```

The AS clause

In HQL, the `AS` clause is used to assign aliases to the classes when you have long queries. The syntax to use the `AS` clause is:

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

The AS clause is optional, so you can also specify the alias directly after the class name as follows:

```
String hql = "FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

The SELECT clause

The SELECT clause gives more control over the result set than the FROM clause. In order to get some specific properties of the object instead of the complete objects, go for the SELECT clause.

The syntax of the SELECT clause is as shown here, where it is just trying to get the name field of the Employee object:

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
return query.list();
```

In this code snippet, E.firstName is the property of the Employee object rather than a field of the Employee table.

The WHERE clause

The WHERE clause is used to narrow the specific objects that are returned from the storage. The syntax of the WHERE clause is:

```
String hql = "FROM Employee E WHERE E.firstName='RAVI'";
Query query = session.createQuery(hql);
List results = query.list();
```

The expected output will be as follows:

```
Hibernate: select employee0_.ID as ID0_, employee0_.DEPARTMENT as
DEPARTMENT0_, employee0_.FIRST_NAME as FIRST3_0_,
employee0_.JOB_TITLE as JOB4_0_, employee0_.LAST_NAME as LAST5_0_,
employee0_.SALARY as SALARY0_ from EMPLOYEE_INFO employee0_ where
employee0_.FIRST_NAME='RAVI'
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]
```

The ORDER BY clause

The ORDER BY clause can be used to sort the results from a HQL query by any property of the objects in the result set, either in the ascending (ASC) or the descending (DESC) order.

The syntax of the ORDER BY clause is as follows:

```
String hql = "FROM Employee E ORDER BY E.firstName DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

The expected output will be as follows:

```
Hibernate: select employee0_.ID as ID0_, employee0_.DEPARTMENT as
DEPARTMENT0_, employee0_.FIRST_NAME as FIRST3_0_,
employee0_.JOB_TITLE as JOB4_0_, employee0_.LAST_NAME as LAST5_0_,
employee0_.SALARY as SALARY0_ from EMPLOYEE_INFO employee0_ order
by employee0_.FIRST_NAME DESC
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer
department=Technology salary=3000]
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]
```

Whenever we need to sort by more than one property in the result set, just add those additional properties to the end of the ORDER BY clause, separated by commas, as follows:

```
String hql = "FROM Employee E ORDER BY E.firstName DESC, E.id
DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

The expected output will be as follows:

```
Hibernate: select employee0_.ID as ID0_, employee0_.DEPARTMENT as
DEPARTMENT0_, employee0_.FIRST_NAME as FIRST3_0_,
employee0_.JOB_TITLE as JOB4_0_, employee0_.LAST_NAME as LAST5_0_,
employee0_.SALARY as SALARY0_ from EMPLOYEE_INFO employee0_ order
by employee0_.FIRST_NAME DESC, employee0_.ID DESC
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer
department=Technology salary=3000]
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]
```

The GROUP BY clause

Hibernate uses the `GROUP BY` clause to pull information from the database and group them based on the value of the attribute and use the result to include an aggregate value.

HQL supports aggregate functions such as `count(*)`, `count(distinct x)`, `max()`, `min()`, `avg()`, and `sum()`. A few are listed here with descriptions:

Function	Description
<code>avg(property name)</code>	This function calculates the average of a property's value
<code>count(property name or *)</code>	This function counts the number of times a given property occurs in the results
<code>max(property name)</code>	This function returns the maximum value from the group
<code>min(property name)</code>	This function returns the minimum value from the group
<code>sum(property name)</code>	This function returns the sum total of the property value

The syntax of the `GROUP BY` clause is as follows:

```
Session session = sessionFactory.openSession();
String hql = "SELECT SUM(E.salary) FROM Employee E GROUP BY
E.firstName";
Query query = session.createQuery(hql);
List<Long> groupList = query.list();
```

The expected output will be as follows:

```
Hibernate: select sum(employee0_.SALARY) as col_0_0_ from
EMPLOYEE_INFO employee0_ group by employee0_.FIRST_NAME

Salary: 3000

Salary: 5000
```

Using the named parameter

Hibernate supports named parameters in HQL queries to accept input from users and you don't have to defend against SQL injection attacks.

The syntax to use named parameters is as shown here:

```
Session session = sessionFactory.openSession();
String hql = "FROM Employee E WHERE E.firstName =
```

```
:employee_firstName";
Query query = session.createQuery(hql);
query.setParameter("employee_firstName", "Shree");
return query.list();
```

The expected output will be as follows:

```
Hibernate: select employee0_.ID as ID0_, employee0_.DEPARTMENT as
DEPARTMENT0_, employee0_.FIRST_NAME as FIRST3_0_,
employee0_.JOB_TITLE as JOB4_0_, employee0_.LAST_NAME as LAST5_0_,
employee0_.SALARY as SALARY0_ from EMPLOYEE_INFO employee0_ where
employee0_.FIRST_NAME=?
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer
department=Technology salary=3000]
```

The UPDATE clause

Hibernate supports bulk updates. The Query interface contains a method named `executeUpdate()` to execute the HQL UPDATE or DELETE statement. The UPDATE clause can be used to update one or more object's properties.

The syntax of the UPDATE clause is as shown here:

```
String hql = "UPDATE Employee E set E.firstName = :name WHERE id =
:employee_id";
Query query = session.createQuery(hql);
query.setParameter("name", "Shashi");
query.setParameter("employee_id", 2);
int result = query.executeUpdate();
System.out.println("Row affected: " + result);
```

The expected output will be as follows:

```
Hibernate: update EMPLOYEE_INFO set FIRST_NAME=? where ID=?

Row affected: 1
```

The DELETE clause

To delete one or more objects, you can use the DELETE clause. The syntax of the DELETE clause is as shown here:

```
String hql = "DELETE from Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 2);
```



```
int result = query.executeUpdate();  
System.out.println("Row affected: " + result);
```

The expected output will be as follows:

```
Hibernate: delete from EMPLOYEE_INFO where ID=?
```

```
Row affected: 1
```

Pagination using Query

HQL supports pagination, where we can construct a paging component in our application. The Query interface supports two methods for pagination:

Method	Description
Query setFirstResult(int startPosition)	This method takes an argument of type int, which represents the result to be retrieved. The row in the result set starts with 0.
Query setMaxResults(int maxResult)	This method takes an argument of type int, and is used to set a limit on the maximum number of objects to be retrieved.

The following code snippet will fetch one row at a time:

```
String hql = "FROM Employee";  
Query query = session.createQuery(hql);  
query.setFirstResult(0);  
query.setMaxResults(1);  
return query.list();
```

The expected output will be as follows:

```
Hibernate: select employee0_.ID as ID0_, employee0_.DEPARTMENT as  
DEPARTMENT0_, employee0_.FIRST_NAME as FIRST3_0_,  
employee0_.JOB_TITLE as JOB4_0_, employee0_.LAST_NAME as LAST5_0_,  
employee0_.SALARY as SALARY0_ from EMPLOYEE_INFO employee0_ limit  
?
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR  
department=TECHNOLOGY salary=5000]
```

Hibernate Criteria Query Language

There is an alternative way provided by Hibernate to manipulate objects and in turn the data available in an RDBMS table. A Java programmer might feel it is easier to use **Hibernate Criteria Query Language (HCQL)** as it supports methods to add criteria on a query.

The Criteria interface

We can build a criteria object using the `Criteria` interface, where we can apply logical conditions and filtration rules. The `Session` interface of Hibernate provides the `createCriteria()` method to create a `Criteria` object that returns an instance of a persistence object's class when your application executes a criteria query.

The following is a list of commonly used methods from the `Criteria` interface:

Method	Description
<code>public Criteria add(Criterion c)</code>	This method is used to add restrictions
<code>public Criteria addOrder(Order o)</code>	This method is used to specify ordering
<code>public Criteria setFirstResult(int firstResult)</code>	This method is used to specify the first number of record to be retrieved
<code>public Criteria setMaxResult(int totalResult)</code>	This method is used to specify the total number of records to be retrieved
<code>public List list()</code>	This method returns the list containing the object
<code>public Criteria setProjection(Projection projection)</code>	This method is used to specify the projection

The following code snippet retrieves all the objects that correspond to the `Employee` class using the criteria query:

```
public List<Employee> getAllEmployees() {
    Session session = sessionFactory.openSession();
    Criteria criteria = session.createCriteria(Employee.class);
    List<Employee> emList = criteria.list();
    return emList;
}
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as  
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE  
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as  
SALARY0_0_ from EMPLOYEE_INFO this_
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR  
department=TECHNOLOGY salary=5000]
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer  
department=Technology salary=3000]
```

Restrictions with Criteria

Restrictive classes provide methods that we can use as Criteria. Let's have a look at a few of them.

The eq method

The eq method will set the equal constraint to a given property.

The syntax of this method is:

```
public static SimpleExpression eq(String propertyName, Object  
value)
```

The following code snippet shows the use of the eq method retrieving all the records of the Employee table whose salary is equal to 5000:

```
Session session = sessionFactory.openSession();  
Criteria criteria = session.createCriteria(Employee.class);  
criteria.add(Restrictions.eq("salary", 5000));  
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as  
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE  
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as  
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.SALARY=?
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR  
department=TECHNOLOGY salary=5000]
```

The gt method

This method sets the greater than constraint to a given property. The syntax of this method is:

```
public static SimpleExpression gt(String propertyName, Object
value)
```

The following code snippet shows the use of the `gt` method retrieving all the records of the Employee table whose ID is greater than 1:

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.gt("id", 1));
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.ID>?
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer
department=Technology salary=3000]
```

The lt method

This method sets the less than constraint to a given property. The syntax of this method is:

```
public static SimpleExpression lt(String propertyName, Object
value)
```

The following code snippet shows the use of the `lt` method retrieving all the records of the Employee table whose `id` is lesser than 3:

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.lt("id", 2));
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as  
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE  
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as  
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.ID<?
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR  
department=TECHNOLOGY salary=5000]
```

The like method

This method sets the like constraint to a given property. The syntax of this method is:

```
public static SimpleExpression like(String propertyName, Object  
value)
```

The following code snippet shows the use of the like method retrieving all the records of the Employee table whose firstName property is like RAVI:

```
Session session = sessionFactory.openSession();  
Criteria criteria = session.createCriteria(Employee.class);  
criteria.add(Restrictions.like("firstName", "RAVI"));  
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as  
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE  
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as  
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.FIRST_NAME like ?
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR  
department=TECHNOLOGY salary=5000]
```

The ilike method

This method sets the ilike constraint to the given property and is case sensitive. The syntax of this method is:

```
public static SimpleExpression ilike(String propertyName, Object  
value)
```

The following code snippet shows the use of the ilike method retrieving all the records of the Employee table whose firstName property is like RAVI:

```
Session session = sessionFactory.openSession();  
Criteria criteria = session.createCriteria(Employee.class);  
criteria.add(Restrictions.ilike("firstName", "RAVI"));  
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.FIRST_NAME ilike ?
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]
```

The between method

This method sets the between constraint. The syntax of this method is:

```
public static Criterion between(String propertyName, Object low,
Object high)
```

The following code snippet shows the use of the between method retrieving all the records of the Employee table whose salary is between 4000 and 5000:

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.between("salary", 4000,5000));
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.SALARY between ? and ?
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]
```

The isNull method

This method sets the isNull constraint to the given property. The syntax of this method is:

```
public static Criterion isNull(String propertyName)
```

The following code snippet shows the use of the isNull method retrieving all the records of the Employee table whose salary is null:

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.isNull("salary"));
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as  
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE  
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as  
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.SALARY is null
```

The isNotNull method

This method sets the `isNotNull` constraint to the given property. The syntax of this method is:

```
public static Criterion isNotNull(String propertyName)
```

The following code snippet shows the use of the `isNotNull` method retrieving all the records of the `Employee` table whose salary is not null:

```
Session session = sessionFactory.openSession();  
Criteria criteria = session.createCriteria(Employee.class);  
criteria.add(Restrictions.isNotNull("salary"));  
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as  
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE  
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as  
SALARY0_0_ from EMPLOYEE_INFO this_ where this_.SALARY is not null
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR  
department=TECHNOLOGY salary=5000]
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer  
department=Technology salary=3000]
```

The And or OR condition

`LogicalExpression` restrictions can be used to create AND or OR conditions as discussed in the following section.

Restrictions.and

The following code snippet shows the `and` condition:

```
Session session = sessionFactory.openSession();  
Criteria criteria = session.createCriteria(Employee.class);  
Criterion salary = Restrictions.eq("salary", 5000);
```

```

Criterion firstName = Restrictions.like("firstName", "RAVI");
LogicalExpression andExp = Restrictions.and(salary, firstName);
criteria.add(andExp);
List<Employee> emList = criteria.list();

```

The expected output will be as follows:

```

Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as
SALARY0_0_ from EMPLOYEE_INFO this_ where (this_.SALARY=? and
this_.FIRST_NAME like ?)

```

```

Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]

```

Restrictions.or

The following code snippet shows the or condition:

```

Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
Criterion jobTitle = Restrictions.eq("jobTitle", "AUTHOR");
Criterion firstName = Restrictions.like("lastName", "Kant");
LogicalExpression orExp = Restrictions.or(jobTitle, firstName);
criteria.add(orExp);
List<Employee> emList = criteria.list();

```

The expected output will be as follows:

```

Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as
SALARY0_0_ from EMPLOYEE_INFO this_ where (this_.JOB_TITLE=? or
this_.LAST_NAME like ?)

```

```

Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR
department=TECHNOLOGY salary=5000]

```

```

Employee [id=2, name=Shree Kant, jobTitle=Software Engineer
department=Technology salary=3000]

```


Pagination using Criteria

HCQL supports pagination, where we can construct a paging component in our application. The Criteria interface supports two methods for pagination:

Method	Description
Public Criteria setFirstResult(int startPosition)	This method takes an argument of type int, which represents the result to be retrieved. The row in the result set starts with 0.
Public Criteria setMaxResults(int maxResult)	This method takes an argument of type int, and is used to set a limit on the maximum number of objects to be retrieved.

The following code snippet will fetch two rows at a time:

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
criteria.setFirstResult(0);
criteria.setMaxResults(2);
List<Employee> emList = criteria.list();
```

Sorting the results

The org.hibernate.criterion.Order class of the Criteria API can be used to sort your results in either ascending or descending order, according to one of the objects' properties.

- public static Order asc(String propertyName): This method applies the ascending order based on a given property
- public static Order desc(String propertyName): This method applies the descending order based on a given property

The following code snippet will order the result by ID in descending order:

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
criteria.addOrder(Order.desc("id"));
List<Employee> emList = criteria.list();
```

The expected output will be as follows:

```
Hibernate: select this_.ID as ID0_0_, this_.DEPARTMENT as
DEPARTMENT0_0_, this_.FIRST_NAME as FIRST3_0_0_, this_.JOB_TITLE
as JOB4_0_0_, this_.LAST_NAME as LAST5_0_0_, this_.SALARY as
SALARY0_0_ from EMPLOYEE_INFO this_ order by this_.ID desc
```

```
Employee [id=2, name=Shree Kant, jobTitle=Software Engineer  
department=Technology salary=3000]
```

```
Employee [id=1, name=RAVI SONI, jobTitle=AUTHOR department=TECHNOLOGY  
salary=5000]
```

Exercise

Q1. What is ORM?

Q2. Explain the basic elements of the Hibernate architecture.

Q3. What is HQL?



The answers to these are provided in *Appendix A, Solution to Exercises*.

Summary

In this chapter, you learned about ORM and understood the various properties of Hibernate in detail. Then we discussed the important elements of the Hibernate architecture. We have successfully configured Hibernate with the Spring application. We have covered a few pieces of Hibernate functionalities and its features. For better understanding of Hibernate, refer to the Hibernate documentation at <http://hibernate.org/>.

You also learned how to use HQL and HCQL to query persistent objects. HQL is the most powerful query language to retrieve objects using different conditions. HCQL provides an object-oriented manner to retrieve persistent objects.

In the next chapter, Spring Security, we will first try to understand what Spring Security is. Then, we will look into the dependencies needed for Spring Security. We will take a look at authentication and authorization in Spring Security. We will take a quick review of Servlet filters in web applications and will understand how Spring Security is dependent on this filter mechanism. Afterthat, we will see the two important aspects of Spring Security: the authentication manager and authentication provider.

5

Spring Security

In the previous chapter, you learned about ORM and understood the various properties of Hibernate. We also learned how to use HQL and HCQL to query persistent objects.

In this chapter, we will first try to understand what Spring Security is. Then, we will look into the dependencies needed for Spring Security. We will take a look at authentication and authorization in Spring Security. Next, we will do a quick review of the Servlet filter in web application and also understand how Spring Security is dependent on this filter mechanism. We will discuss how to secure web applications using filters along with the Spring interceptor and filter concepts in Spring Security. Then, we will see the two important aspects of Spring Security, that is, the authentication manager and authentication provider. We will also see different ways of logging into web applications, such as HTTP basic authentication, form-based login services, anonymous login, and also the Remember Me support in Spring Security. We will also discuss authenticating and authorization against databases. Then, we will implement method-level security.

The list of topics covered in this chapter is as follows:

- Introduction to Spring Security
- Review on Servlet filters
- Security use case
- Spring Security configuration
- Securing web application's URL access
- Logging into web application
- Users authentication

- Method-level security
- Developing an application using Spring MVC, Hibernate, and Security

What is Spring Security?

Security for a web application is nothing but protecting resources and allowing only specific users to access it. Spring Security shouldn't be assumed as a firewall, a proxy server, intrusion detection, JVM security, or anything similar. Spring Security is basically made for the Java EE Enterprise software application and is primarily targeted towards Spring-framework-based web applications.

The Spring Security framework initially started as Acegi Security Framework, which was later adopted by Spring as its subproject Spring Security. The Spring Security framework is a de facto standard to secure Spring-based applications. The Spring Security framework provides security services for enterprise Java software applications by handling authentication and authorization. Spring Security handles authentication and authorization at both the web request level and the method invocation level. Spring Security is a highly customizable and powerful authentication and access control framework.

Major operations

The two major operations provided by Spring Security are authentication and authorization.

- **Authentication:** This is the process of assuring that the user is the one that the user claims to be. Authentication is a combination of identification and verification. Identification can be performed in a number of ways. For example, through a username and password that can be stored in a database, LDAP, or CAS (single sign-on protocol). Spring Security provides a password encoder interface to make sure that the user's password is hashed.
- **Authorization:** This provides access control to an authenticated user. Authorization is the process of assuring that the authenticated user is allowed access only to those resources that they are authorized to use. Let's take an example of the HR Payroll application, where some parts of the application have access to HR and to some other parts all the employees have access. The access rights given to the user of the system will determine the access rules.

In web-based applications, this is often done through URL-based security and is implemented using filters that play a primary role in securing the Spring web application.

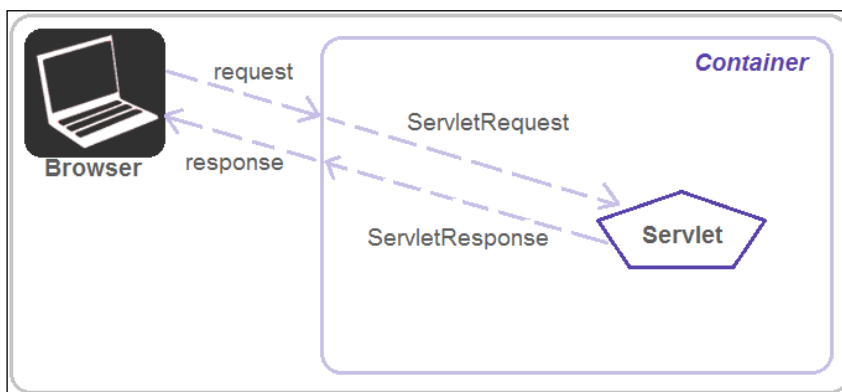
Sometimes, URL-based security is not enough for web applications as URLs can be manipulated and have relative pass. Let's take an example of `HrPayrollSystem`, where the HR and manager are involved, and there is an employees list page. On this employees list page, there is a **Delete** button for each employee. The **Delete** button contains a hyperlink for a `delete` method call in the controller class. This button appears for HR but it is hidden for managers. Even though the manager doesn't see the **Delete** button, the `delete` method can be called by altering the URL in the browser. This results in the delete operation by the manager, which shouldn't have happened.

So, Spring Security also provides method-level security. The authorized user will only be able to invoke those methods which he is granted for.

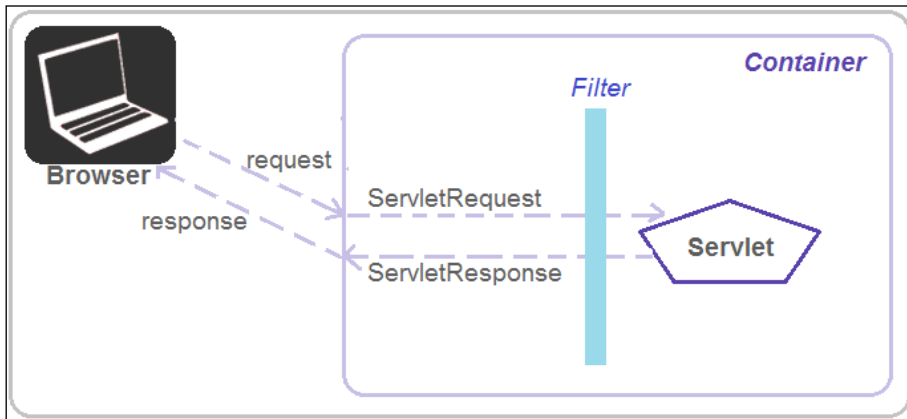
Servlet filters review

Spring Security is developed on top of the Spring Framework and uses the filters concept in the Servlet engine. Filters are like Servlet; they come into action when any request comes to Servlet and can decide whether the request should be forwarded to Servlet or not. Spring Security registers a single `javax.servlet.Filter`, that is, the `DelegatingFilterProxy`.

Before starting with Spring Security, let's quickly recall what **Servlet** filters are. In the following figure, a user enters the URL in the browser. The request comes to the container and then to **Servlet** after referring to `web.xml` for Servlet mapping with respecting URL. After processing the request, the request goes back to the user.



A **Filter** is present between **Servlet** and **Container**. It intercepts the requests and responses to and from Servlet and can pre-process and post-process, as shown in the following diagram:



In the `web.xml` file, you'll find the following code

```
<filter>
  <filter-name>filterA</filter-name>
  <filter-class>FilterA</filter-class>
</filter>
<filter-mapping>
  <filter-name>filterA</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

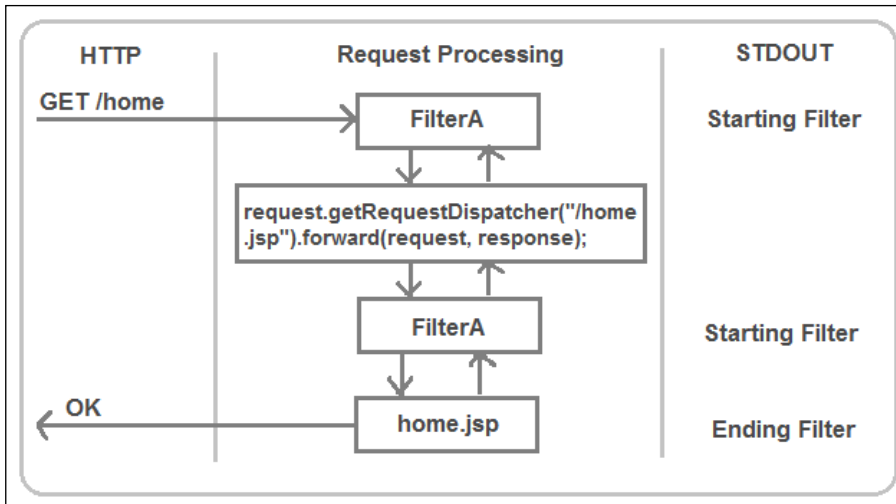
In the preceding code snippet, we have mapped `filterA` to all URLs. Now, in the `FilterA.java` class, you'll find the following code

```
public void doFilter(ServletRequest request, ServletResponse
response, FilterChain filterChain)
{
    // do something before filter
    System.out.println("Starting Filter");

    // run rest of the application
    filterChain.doFilter(request, response);

    // cleanup
    System.out.println("Ending Filter");
}
```

Now, we have the code for `FilterA`. First, it invokes a message before the rest of the applications run. Then, it runs the rest of the application. Lastly, it prints a message again. From the following diagram, let's understand how requests gets impacted by this filter



As shown in the preceding diagram, when we make a request to our application using HTTP GET /home URL, the Servlet container recognizes the `filterA` intercepts this URL. The container invokes the `doFilter()` method of the `FilterA` class. As soon as the `doFilter()` method is invoked, it prints the message `Starting Filter`. Then, `filterA` invokes the `filterChain`, and then `home.jsp` is invoked. Next, it returns to the `filterChain`.

Filters can be used for the following operations:

- Blocking access to a resource based on user identity or role membership
- Auditing incoming requests
- Comparing the response data stream
- Transforming the response
- Measuring and logging Servlet performance

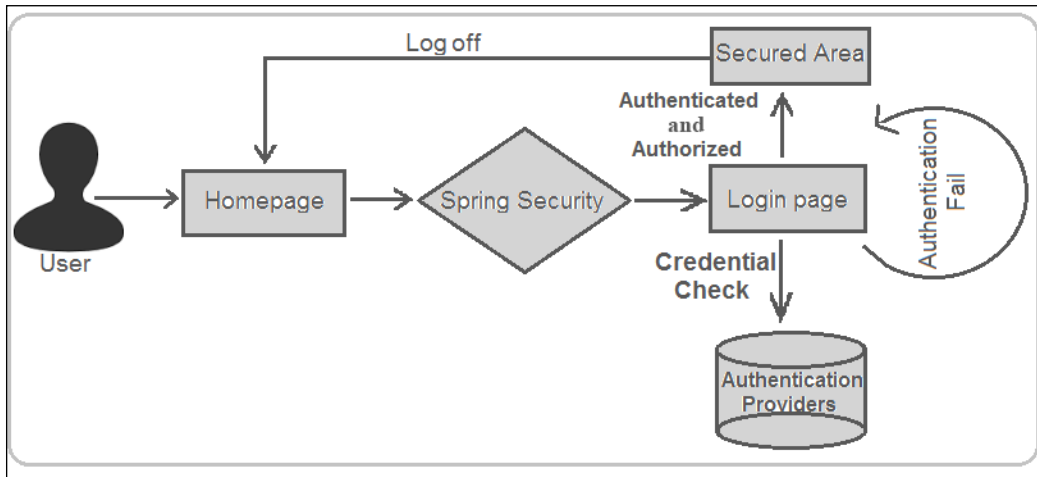
Spring Security is dependent on this filter mechanism. So, before reaching out to Servlet to perform some business logic, some security can be performed using the filters

Security use case

The use case we will use for all our examples is as follows:

1. The user reaches the application or homepage of the application and clicks on a secure link (for example, login).
2. The moment the user clicks on the secured link, Spring Security brings the login page.
3. The login page will perform a credential check from the authentication provider; this can be plain-text, database, or similar.
4. An authentication failure happens if wrong credentials are given by the user; otherwise, the user will be allowed to the secured area.
5. When the user clicks on logout, they will be directed to the homepage.

The following diagram illustrates the preceding steps:



Spring Security configuration

To add Spring Security to our Spring web application, we need to perform a basic Spring Security setup. To do this, follow these steps:

1. Add Spring JARs or Spring Security dependencies.
2. Update `web.xml` with `springSecurityFilterChain`.
3. Create a Spring Security configuration file

Spring Security setup

We can either download and add Spring Security JARs to classpath or we can provide dependencies to Maven.

Adding JARs to the classpath

There are three important JARs that we need for Spring Security. These can be downloaded from the Spring website and are as follows (the version should match other Spring JARs used in the project):

- `spring-security-config-3.X.X.RELEASE.jar`: This contains support for Spring Security's XML namespace
- `spring-security-core-3.X.X.RELEASE.jar`: This provides the essential Spring Security library
- `spring-security-web-3.X.X.RELEASE.jar`: This provides Spring Security's filter-based web security support

If we have developed a Maven application, then we need to update `pom.xml`.

Spring Security dependencies – pom.xml

Update dependencies to Maven. We have `spring-security-core`, `spring-security-web`, and `spring-security-config`:

```
<properties>
  <spring.security.version>3.1.4.RELEASE</spring.security.version>
</properties>

<!-- Spring Security -->
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>${spring.security.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>${spring.security.version}</version>
</dependency>

<dependency>
```

```
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-config</artifactId>
<version>${spring.security.version}</version>
</dependency>
```

Namespace configuration

In the Spring configuration file, we need to add one more entry to schema, which is related to Spring Security and the corresponding schemaLocation and their xsd, which lives in Spring JARs. The security prefixed elements go here

In the `SpringDispatcher-servlet.xml` file, you'll find the following code

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:security="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-
3.1.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-
3.1.xsd">
</beans>
```

Securing web application's URL access

`HttpServletRequest` is the starting point of a Java web application. To configure web security, you need to set up a filter that provides various security features. To enable Spring Security, add filter and their mapping in the `web.xml` file

The first step – web.xml

The first step is to configure `DelegatingFilterProxy` instance in `web.xml` while securing the web application's URL access with Spring Security.

In the `web.xml` file, you'll find the following code

```
<!--Spring Security -->
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.
DelegatingFilterProxy</filter-
class>
</filter>
```

```

<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

The `DelegatingFilterProxy` filter class, which is a special servlet filter, doesn't do much by itself. It delegates the control to an implementation of `javax.servlet.Filter`, which is registered as a special bean with ID `springSecurityFilterChain` in Spring application context. In the earlier code snippet, we added `/*`, which will map to all the HTTP requests and go to this `springSecurityFilterChain`.

In the preceding code snippet, we declared the URL pattern `/*`, which requires some level of granted authority and prevents other users without authority from accessing the resources behind those URLs.

Separating security configurations

If we are planning to separate the entire security specific configuration into separate configuration file name `Spring-Security.xml`, we must change the security namespace to be the primary namespace for that file. Here, there are no security prefixed elements

In the `Spring-Security.xml` file, you'll find the following code

```

<beans:beans
xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-
3.1.xsd">

  <http auto-config="true">
    <intercept-url pattern='/employeeList'
access='ROLE_USER,ROLE_ADMIN' />
    <intercept-url pattern='/employeeAdd' access='ROLE_USER'
/>
    <intercept-url pattern='/employeeDelete'
access='ROLE_ADMIN' />

  </http>

```

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="admin" password="adminpassword"
authorities="ROLE_ADMIN" />
      <user name="ravisoni" password="mypassword"
authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>

</beans:beans>
```

The preceding configuration file has been divided into two major sections, as show in previous code snippet:

- The first section includes `<http>` tag and `<intercept-url>` tag; these define what you want to secure
- The second section includes the `<authentication-manager>`, `<authentication-provider>`, and `<user-service>` tags; these define how you want to secure

Web security is enabled using the `<http>` tag. This tag is the container element for the HTTP security configuration. To define the Spring Security configuration for HTTP requests, we must first define the `<http>` tag, which automatically sets up `FilterChainProxy`. The `auto-config=true` attribute automatically configures the basic Spring Security Services that a web application needs. This can be fine-tuned with the corresponding subelements in it.

The `<intercept-url>` element is defined inside the `<http>` configuration element. It restricts access to specific URLs. The `<intercept-url>` tag defines the URL pattern and set of access attributes that are required to access URLs. It is mandatory to include a wildcard at the end of the URL pattern, and failing to do so will allow a hacker to skip the security check by appending arbitrary request parameter. The access attributes decide if the user can access the URLs. In most cases, access attributes are defined in terms of roles. In the previous code snippet, users with the `ROLE_USER` role are able to access the `/employeeList` and `/employeeAdd` URLs. However, to delete an employee via the `/employeeDelete` URL, a user must have the `ROLE_ADMIN` role.

The `<authentication-manager>` tag used to process authentication information. The `<authentication-provider>` tag is nested inside the `<authentication-manager>` tag, and used to define credential information and the roles that will be granted to this user. In the preceding code snippet, inside the `<authentication-manager>` tag, we have provided the `<authentication-provider>` tag, which specifies a text-based user ID and password

Logging into web application

Users can log into a web application using multiple ways supported by Spring Security:

- **HTTP basic authentication:** These processes the basic credentials presented in the header of the HTTP request. HTTP basic authentication is generally used with stateless clients which pass their credentials on each request.
- **Form-based login service:** This provides the default login form page for users to log into the web application.
- **Logout service:** This allows users to log out of this application.
- **Anonymous login:** This grants authority to an anonymous user like normal user.
- **Remember Me support:** This remembers a user's identity across multiple browser sessions.

First, we will disable the HTTP autoconfiguration by removing the `auto-config` attribute from the `<http>` tag to better understand the different login mechanisms in isolation:

```
<http>
    <intercept-url pattern='/employeeList'
access='ROLE_USER,ROLE_ADMIN ' />
    <intercept-url pattern='/employeeAdd' access='ROLE_USER'
/>
    <intercept-url pattern='/employeeDelete'
access='ROLE_ADMIN' />

</http>
```

HTTP basic authentication

The HTTP basic authentication in Spring Security can be configured by using the `<http-basic/>` element. Here, the browser will display a login dialog for user authentication:

```
<beans:beans
xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-
3.1.xsd">

    <http>
        <intercept-url pattern='/employeeList'
access='ROLE_USER,ROLE_ADMIN' />
        <intercept-url pattern='/employeeAdd' access='ROLE_USER'
/>
        <intercept-url pattern='/employeeDelete'
access='ROLE_ADMIN' />

        <!-- Adds Support for basic authentication -->
        <http-basic/>

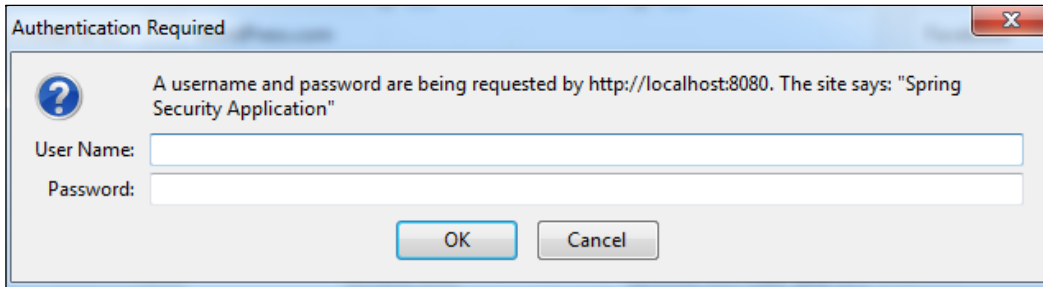
    </http>

    <authentication-manager>
        <authentication-provider>
            <user-service>
                <user name="admin" password="adminpassword"
authorities="ROLE_ADMIN" />
                <user name="ravisoni" password="mypassword"
authorities="ROLE_USER" />
            </user-service>
        </authentication-provider>
    </authentication-manager>

</beans:beans>
```

The interesting thing with HTTP basic authentication is that we don't have to create any login page. The browser will present a login box before the user on our behalf. As each request contains user authentication information that is the same as the HTTP stateless mechanism, we don't need to maintain session.

When we try to access a secured URL in our web application, the browser will open an authentication dialog box automatically for a username and password:



Form-based login service

Spring Security supports form-based login service by providing the default login form page for users to input their login details. The `<form-login>` element defines the support for the login form, as shown in the following code snippet. By default, a login form, which will map to the `/spring_security_login` URL, will automatically be created by Spring Security, as shown here:

```
<http>
. . .
<!-- Adds Support for basic authentication -->
<form-login />
</http>
```

We can also create our own custom login page (`login.jsp`) in the root directory of the web application. This should not go inside `WEB-INF` as it prevents users from accessing it directly. The form action URL in `login.jsp` will take the `j_spring_security_check` value; this is the URL where the form will be posted to trigger the authentication process, and `j_username` is used as the username and `j_password` is used as the password, as shown in the following code snippet:

```
<html>
<head>
<title>Login</title>
</head>

<body>
<form action="j_spring_security_check" method='POST'>
  <table>
    <tr>
      <td>UserName:</td>
```



```
        <td><input type='text' name='j_username' value=''></td>
    </tr>
    <tr>
        <td>Password:</td>
        <td><input type='password' name='j_password' /></td>
    </tr>
    <tr>
        <td>Remember me:</td>
        <td><input type='checkbox' name='_spring_security_
remember_me' /></td>
    </tr>
    <tr>
        <td><input name="submit" type="submit" value="submit"
/></td>
    </tr>
</table>
</form>
</body>
</html>
```

While referring to the custom login page for Spring Security, we need to specify its URL in the `login-page` attribute of `<form-login>`. As shown in following code snippet, `<form-login login-page="/login" authentication-failure-url="/loginfailed" default-target-url="/employeeList" />` defines that when the login button is clicked, it should be navigated to `/login`. The default target URL is defined as `/employeeList`; this means when a user is authenticated, this URL hits by default. When an authentication failure happens, it should navigate to `/loginfailed`:

```
<http>
    . . .
    <form-login login-page="/login" authentication-failure-
url="/loginfailed" default-target-url="/employeeList" />

</http>
```

Logout service

The logout service handles logout requests and is configured via the `<logout>` element. In Spring Security, by default, it is mapped to the `/j_spring_security_logout` URL, and it redirects the user to the context path's root when the logout is successful:

```
<http>
. . .
<logout />
</http>
```

We can provide the logout link in our page by referring the URL ` Logout `.

We can also configure log out so that the user is redirected to another URL after the logout is successful, as shown in the following code snippet:

```
<http>
. . .
<logout logout-success-url="/login" />
</http>
```

Anonymous login

The `<anonymous>` element is used to configure anonymous login service, where the username and authority of the anonymous user can be configured

```
<http>
    <intercept-url pattern='/employeeList' access='ROLE_
USER,ROLE_ADMIN,ROLE_GUEST' />
    <intercept-url pattern='/employeeAdd' access='ROLE_USER'
/>
    <intercept-url pattern='/employeeDelete'
access='ROLE_ADMIN' />
. . .
    <anonymous username='guest' granted-
authority='ROLE_GUEST' />
</http>
```

Remember Me support

The `<remember-me />` element is used to configure the Remember Me support in Spring Security. By default, it encodes authentication information and the Remember Me expiration time along with private key as a token. It stores this to the user's browser cookie. The next time a user accesses the same application, they can be log in automatically using the token:

```
http>
. . .
<remember-me />
</http>
```

Users authentication

While users log into applications to access secure resources, the user's principle needs to be authenticated and authorized. The authentication provider helps in authenticating users in Spring Security. If a user is successfully authenticated by the authentication provider, then only the user will be able to log into the web application, otherwise, the user will not be able to log into the application.

There are multiples of ways supported by Spring Security to authenticate users, such as a built-in provider with a built-in XML element, or authenticate user against a user repository (relational database or LDAP repository) storing user details. Spring Security also supports algorithms (MD5 and SHA) for password encryption.

Users authentication with in-memory definitions

If there are only few users for your application with infrequent modification in their details, then you can define user details in Spring Security's configuration file instead of extracting information from the persistence engine, so that their details are loaded into your application's memory, as shown here:

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="admin" password="adminpassword"
authorities="ROLE_ADMIN" />
      <user name="ravisoni" password="mypassword"
authorities="ROLE_USER" />
      <user name="user" password="mypassword"
disabled="true" authorities="ROLE_USER" />
```

```

        </user-service>
    </authentication-provider>
</authentication-manager>

```

The user's details can be defined in `<user-service>` with multiple `<user>` elements. For each user, a username, password, disabled status, and a set of granted authority can be specified, as shown in the previous code snippet. The disabled user indicates that the user cannot log into system anymore.

The user details can also be externalized by keeping them in the properties file (for instance, `/WEB-INF/usersinfo.properties`):

```

<authentication-manager>
    <authentication-provider>
        <user-service properties="/WEB-
INF/usersinfo.properties" />
    </authentication-provider>
</authentication-manager>

```

Next, we will see the specified properties file containing user details in the form of properties, where each property represents the user's details. In this property file, the key of the property represents the username, while the property value is divided into several parts separated by commas. The first part represents the password and the second part represents the user's enable status (this is optional with the default status is enabled), and the remaining parts represent authority granted to the user.

The `/WEB-INF/usersinfo.properties` file is as follows:

```

admin=adminpassword,ROLE_ADMIN
ravisoni=mypassword,ROLE_USER
user=mypassword,disabled,ROLE_USER

```

Users authentication against database

If you have a huge list of users in your application and you frequently modify their details, you should consider storing the user details in a database for easy maintenance. Spring Security provides built-in support to query user details from the database.

In order to perform authentication against database, tables need to be created to store users and their roles details. Refer to <http://docs.spring.io/spring-security/site/docs/3.2.3.RELEASE/reference/htmlsingle/#user-schema> for more details on user schema.

The `USER_AUTHENTICATION` table is used to authenticate the user and contains the following columns.

The script is as follows:

```
CREATE TABLE USER_AUTHENTICATION (  
    USERNAME VARCHAR(45) NOT NULL ,  
    PASSWORD VARCHAR(45) NOT NULL ,  
    ENABLED SMALLINT NOT NULL DEFAULT 1,  
    PRIMARY KEY (USERNAME)  
);
```

The table structure is as follows:

Username	Password	Enabled
admin	adminpassword	1
ravisoni	mypassword	1
user	mypassword	0

The `USER_ AUTHORIZATION` table is used to authorize the user and contains the following columns.

The script is as follows:

```
CREATE TABLE USER_ AUTHORIZATION (  
    USERNAME VARCHAR(45) NOT NULL,  
    AUTHORITY VARCHAR(45) NOT NULL,  
    FOREIGN KEY (USERNAME) REFERENCES USERS  
);
```

The table is as follows:

Username	Authority
admin	ROLE_ADMIN
ravisoni	ROLE_USER
user	ROLE_USER

Now, `dataSource` has to be declared in the Spring configuration file to allow Spring Security to access these tables, which will help in creating a connection to the database, as shown here:

```
<bean id="dataSource"  
    class="org.springframework.jdbc.datasource.DriverManager  
    DataSource">
```

```

        <property name="driverClassName"
value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
value="jdbc:derby://localhost:1527/test;create=true" />
        <property name="username" value="root" />
        <property name="password" value="password" />
    </bean>

```

Then, configure the authentication provider using the `<jdbc-user-service>` element that queries this database. Specify the query statement to get the user's information and authority in the `user-by-username-query` and `authorities-by-username-query` attributes, as follows:

```

<authentication-manager>
    <authentication-provider>
        <jdbc-user-service data-source-ref="dataSource"
            user-by-username-query=
                "select username, password, enabled from
user_authentication where username=?"
            authorities-by-username-query=
                "select username, authority from
user_authorization where username =? "
        />
    </authentication-provider>
</authentication-manager>

```

Encrypting passwords

Spring Security supports some hashing algorithms, such as MD5 (`Md5PasswordEncoder`), SHA (`ShaPasswordEncoder`), and BCrypt (`BCryptPasswordEncoder`) for password encryption.

To enable the password encoder, use the `<password-encoder/>` element and set the `hash` attribute, as follows:

```

<authentication-manager>
    <authentication-provider>
        <password-encoder hash="md5" />
        <jdbc-user-service data-source-ref="dataSource"
            . . .
    </authentication-provider>
</authentication-manager>

```

Method-level security

This is an alternative to securing URL access in the web layer. Sometimes, it is also required to secure method invocation in the service layer by enforcing fine-grained security control on methods. This is because, sometimes, it's easier to control it on particular methods than filtering by address, which can be called by typing. We can secure method invocation using Spring Security in a declarative way. We can annotate methods declaration in a bean interface or its implementation class with `@Secured` annotation and specify the access attributes as its value whose type is `String[]`, and enable security for these annotated methods by adding `<global-method-security>` in `Spring-Security.xml` file. This can be done as follows

```
<beans:beans
xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-
3.1.xsd">

    <!-- To allow standards-based @Secured annotation, enable
secured-annotations -->
    <global-method-security secured-annotations="enabled" />

    <http
        . . .
        . . .

</beans>
```

The `global-method-security` namespace is configured along with its `secured-annotations="enabled"` attribute to enable annotation-based security. And annotate methods with `@Secured` annotation to allow method access for one or more than one role:

```
public interface EmployeeService {

    @Secured("ROLE_USER", "ROLE_GUEST")
    public List<employee> employeeList();

    @Secured("ROLE_USER", "ROLE_ADMIN")
    public Person employeeAdd(Employee employee);

    @Secured("ROLE_ADMIN")
```

```
public Person employeeDelete(int employeeId);

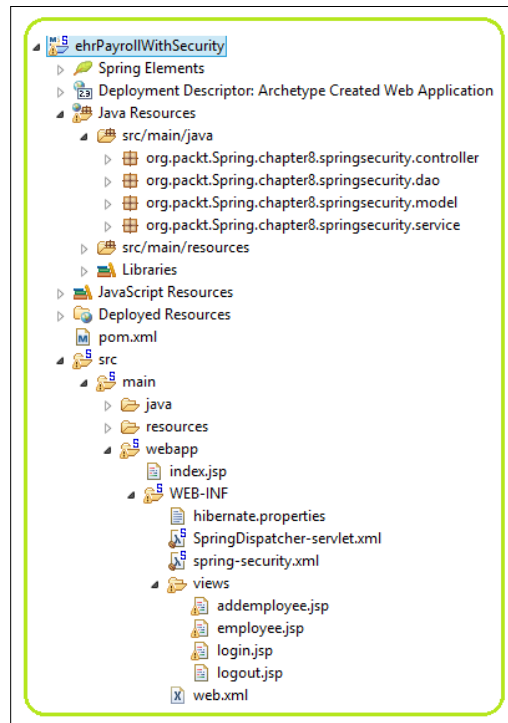
}
```

Let's get down to business

In this section, we will develop an application using Spring MVC, Hibernate, and Spring Security. Here, we have a custom login page, logout page, employee page (to list employees), and add employee page (to add employees), which is secured by the Spring Framework. A user can log into the application using the custom login page and view the secured page based on the authentication and authorization. A user will be redirected to the custom login page on any authentication failure along with the error message, which describes the reason for failure. User will be logged out from the application on clicking on the logout link and redirected to the logout page.

Project structure

The overall project structure is as follows:



In the `pom.xml` file, you'll find the following code

A list of all required dependencies are listed here in `pom.xml`. To get Spring Security features, you need to add `spring-security-core`, `spring-security-web`, and `spring-security-config` to the `pom.xml` file

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.packt.Spring.chapter8.springsecurity</groupId>
  <artifactId>ehrPayrollWithSecurity</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>ehrPayrollWithSecurity Maven Webapp</name>
  <url>http://maven.apache.org</url>
```

Here, the properties specify the versions used:

```
<properties>
  <spring.version>4.1.3.RELEASE</spring.version>
  <security.version>4.0.0.CI-SNAPSHOT</security.version>
  <hibernate.version>4.2.11.Final</hibernate.version>
  <org.aspectj-version>1.7.4</org.aspectj-version>
</properties>
```

Here are the dependencies for all the JARs:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <!-- Spring -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
```

```

    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- Spring transaction -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- Spring Security -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>${security.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>${security.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${security.version}</version>
</dependency>
<!-- Spring ORM -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- AspectJ -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${org.aspectj-version}</version>
</dependency>
<!-- Hibernate ORM framework dependencies -->
<dependency>
    <groupId>org.hibernate</groupId>

```

```
        <artifactId>hibernate-core</artifactId>
        <version>${hibernate.version}</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>${hibernate.version}</version>
    </dependency>
    <!-- Java Servlet and JSP dependencies (for compilation only) -
->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>3.0.1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.1</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSTL dependency -->
    <dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <!-- Apache Commons DBCP dependency (for database connection
pooling) -->
    <dependency>
        <groupId>commons-dbcp</groupId>
        <artifactId>commons-dbcp</artifactId>
        <version>1.4</version>
    </dependency>
    <!-- postgresql Connector Java dependency (JDBC driver for
postgresql) -->
    <dependency>
        <groupId>postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>9.0-801.jdbc4</version>
    </dependency>
    <!-- logging -->
    <dependency>
```

```

        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.4.2</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
<build>
    <finalName>ehrPayrollWithSecurity</finalName>
</build>
</project>

```

Adding filters to web.xml

Add filters to `web.xml`, where all incoming requests will be handled by Spring Security. The Spring Security JAR contains `DelegatingFilterProxy`, which delegates control to a filter chaining in the Spring Security internals. The bean name should be `springSecurityFilterChain`.

In the `web.xml` file, you'll find the following code

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">
    <display-name>Archetype Created Web Application</display-name>
    <servlet>
        <servlet-name>SpringDispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>SpringDispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <listener>

```

```
<listener-class>org.springframework.web.context.  
ContextLoaderListener</  
listener-class>  
</listener>  
<context-param>  
  <param-name>contextConfigLocation</param-name>  
  <param-value>  
    /WEB-INF/SpringDispatcher-servlet.xml,  
    /WEB-INF/spring-security.xml  
  </param-value>  
</context-param>  
<!-- Spring Security -->  
<filter>  
  <filter-name>springSecurityFilterChain</filter-name>  
  <filter-class>org.springframework.web.filter.  
DelegatingFilterProxy</filter  
-class>  
</filter>  
<filter-mapping>  
  <filter-name>springSecurityFilterChain</filter-name>  
  <url-pattern>/*</url-pattern>  
</filter-mapping>  
</web-app>
```

Resolving your view

To resolve the view, view resolver has been added to the `SpringDispatcher-servlet.xml` configuration file. Also `dataSource`, `sessionFactory`, and `transactionManager` have been defined here

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xmlns:aop="http://www.springframework.org/schema/aop"  
  xmlns:context="http://www.springframework.org/schema/context"  
  xmlns:mvc="http://www.springframework.org/schema/mvc"  
  xmlns:tx="http://www.springframework.org/schema/tx"  
  xsi:schemaLocation="http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans.xsd  
  http://www.springframework.org/schema/aop  
http://www.springframework.org/schema/aop/spring-aop-4.1.xsd  
  http://www.springframework.org/schema/context  
http://www.springframework.org/schema/context/spring-context-  
4.1.xsd
```

```

    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">
    <context:component-scan base-
package="org.packt.Spring.chapter8.springsecurity" />
    <context:property-placeholder location="/WEB-
INF/hibernate.properties" />
    <bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource
">
    <property name="driverClassName"
value="{jdbc.driverClassName}" />
    <property name="url" value="{jdbc.url}" />
    <property name="username" value="{jdbc.username}" />
    <property name="password" value="{jdbc.password}" />
    </bean>
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate4.
LocalSessionFactoryBean"
>
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses"
value="org.packt.Spring.chapter8.springsecurity.model.
Employee" />
    <property name="hibernateProperties">
    <props>
    <prop key="hibernate.dialect">{hibernate.dialect}</prop>
    <prop key="hibernate.show_sql">{hibernate.show_sql}</
prop>
    </props>
    </property>
    </bean>
    <bean id="transactionManager"
class="org.springframework.orm.hibernate4.HibernateTransaction
Manager">
    <property name="sessionFactory" ref="sessionFactory" />
    </bean>
    <tx:annotation-driven transaction-manager="transactionManager"
/>
    <bean
class="org.springframework.web.servlet.view.InternalResource
ViewResolver">
    <property name="prefix">

```

```
        <value>/WEB-INF/views/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
</beans>
```

Let's add a custom login

We have defined a role called `ROLE_ADMIN`. We have defined credentials for this role. Also, we have mapped URLs with roles that will be handled by Spring Security. To provide custom login form, add `<form:login>` in this file. When the user tries to access a secured resource, a custom login page will be served.

In the `security-config.xml` file, you'll find the following code

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/security"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-
3.1.xsd">
    <http auto-config="true">
        <intercept-url pattern="/employee/*" access="ROLE_ADMIN" />
        <form-login login-processing-url="/login" login-page="/
loginPage"
            username-parameter="username" password-
parameter="password"
            default-target-url="/employee/listemployee"
            authentication-failure-url="/loginPage?auth=fail" />
        <logout logout-url="/logout" logout-success-
url="/logoutPage" />
    </http>
    <authentication-manager>
        <authentication-provider>
            <user-service>
                <user name="ravi" password="ravi@123" authorities="ROLE_
ADMIN" />
            </user-service>
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

Mapping your login requests

The `LoginController` class contains two methods, namely `logoutPage` and `loginPage`, with request mapping. The `/loginPage` redirects the user to the login page and the `/logoutpage` redirects the user to the logout page:

```
package org.packt.Spring.chapter8.springsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class LoginController {

    @RequestMapping(value = "/logoutPage", method =
RequestMethod.GET)
    public String logoutPage() {
        return "logout";
    }

    @RequestMapping(value = "/loginPage", method =
RequestMethod.GET)
    public String loginPage() {
        return "login";
    }
}
```

Obtaining the employee list

This controller class has the `listEmployee()`, `addEmployee()`, and `deleteEmployee()` methods. In the `EmployeeController.java` file, you'll find the following code

```
package org.packt.Spring.chapter8.springsecurity.controller;

import org.packt.Spring.chapter8.springsecurity.model.Employee;
import org.packt.Spring.chapter8.springsecurity.service.
EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.ModelAttribute;
```



```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("/employee")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @RequestMapping(value = "/listemployee", method =
RequestMethod.GET)
    public String listEmployees(ModelMap model) {
        model.addAttribute("employeesList",
employeeService.listEmployee());
        return "employee";
    }

    @RequestMapping(value = "/addemployee", method =
RequestMethod.GET)
    public ModelAndView addEmployee(ModelMap model) {
        return new ModelAndView("addemployee", "command", new
Employee());
    }

    @RequestMapping(value = "/updateemployee", method =
RequestMethod.POST)
    public String updateEmployee(
        @ModelAttribute("employeeForm") Employee employee,
ModelMap model) {
        this.employeeService.insertEmployee(employee);
        model.addAttribute("employeesList",
employeeService.listEmployee());
        return "employee";
    }

    @RequestMapping(value = "/delete/{empId}", method =
RequestMethod.GET)
    public String deleteEmployee(@PathVariable("empId") Integer
empId,
        ModelMap model) {
        this.employeeService.deleteEmployee(empId);
    }
}
```

```

        model.addAttribute("employeesList",
employeeService.listEmployee());
        return "employee";
    }
}

```

Let's see some credentials

This login page provides an input box to accept credentials from the user. In the `login.jsp` file, you'll find the following code

```

<%@ taglib uri='http://java.sun.com/jsp/jstl/core' prefix='c'%>
<html>
<head>
<title>Login Page</title>
</head>
<body>
    <h2 style="color: orange">Login to eHR Payroll</h2>
    <c:if test="${'fail' eq param.auth}">
        <div style="color:red">
            Login Failed!!!<br />
            Reason : ${sessionScope["SPRING_SECURITY_LAST_
EXCEPTION"].message}
        </div>
    </c:if>
    <form action="${pageContext.request.contextPath}/login"
method="post">
        <table frame="box" cellpadding="0" cellspacing="6">
            <tr>
                <td>Username:</td>
                <td><input type='text' name='username' /></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><input type='password' name='password'></td>
            </tr>
            <tr>
                <td colspan='2'><input name="submit" type="submit"
value="Submit"></td>
            </tr>
        </table>
    </form>
</body>
</html>

```

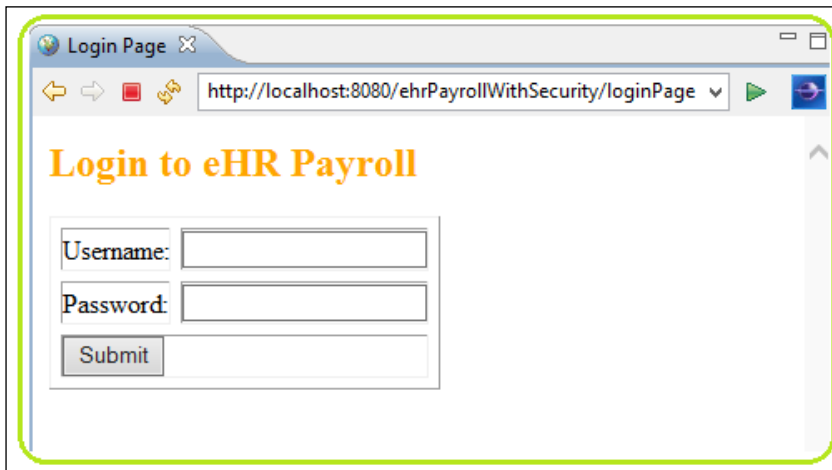
Time to log out

This logout page reflects that the user has been logged out from the application. In the `logout.jsp` file, you'll find the following code

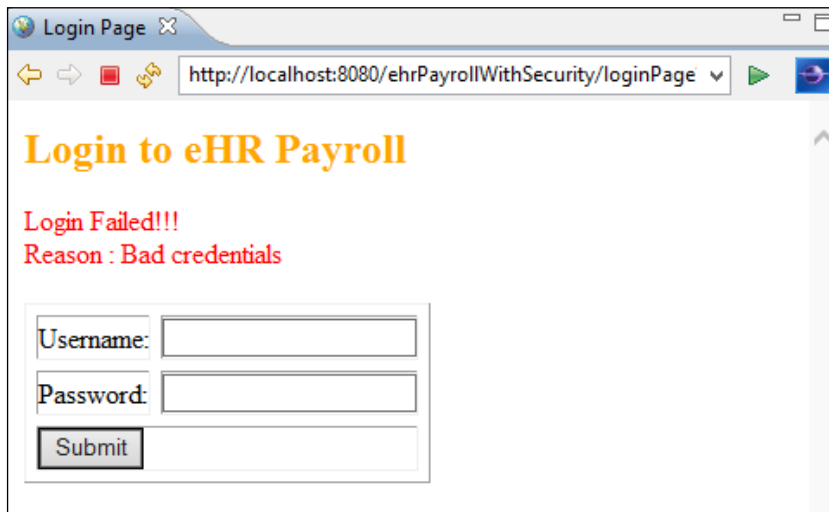
```
<html>
<title>Logout Page</title>
<body>
    <h2>You have been successfully logged out.</h2>
    <a href="{pageContext.request.contextPath}/employee/
listemployee">
Login to eHR Payroll</a>
</body>
</html>
```

Running the application

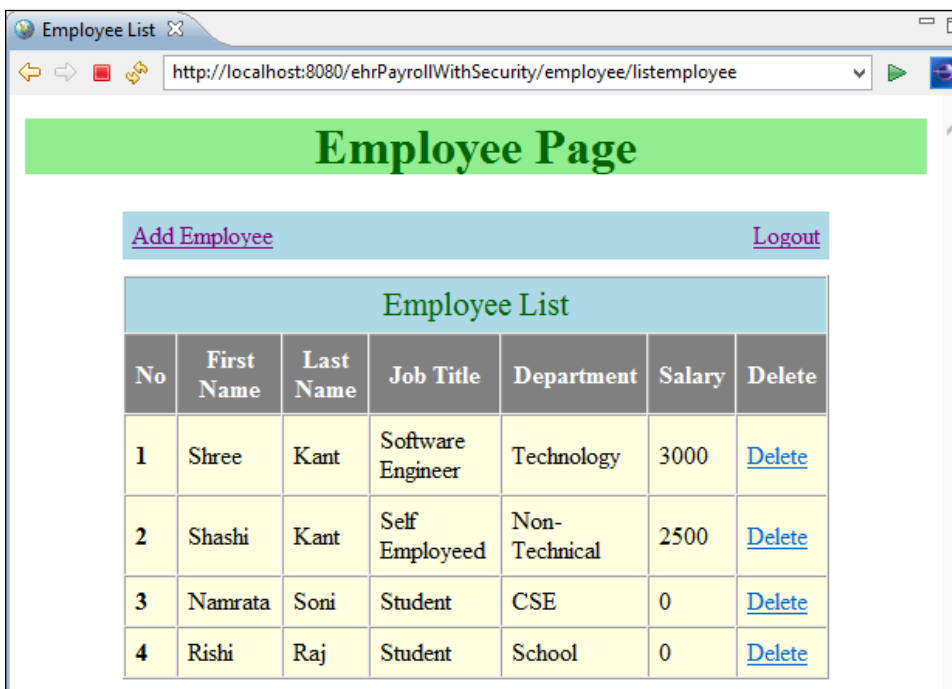
Once you deploy the web application after starting the server, open the URL `http://localhost:8080/ehrPayrollWithSecurity/loginPage` a custom login page will appear:



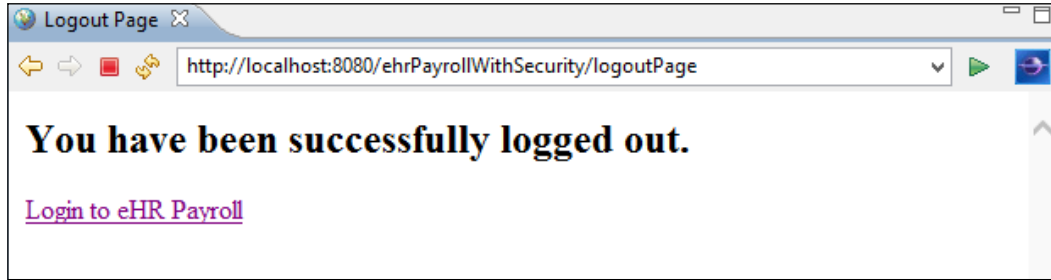
If you enter the wrong credentials, the following error will appear:



If you enter the correct credentials, you will be navigated to the `listEmployee` page:



Clicking on **Logout** will navigate you to the logout page, as shown here:




Exercise

Q1. What is Spring Security?

Q2. What is authentication and authorization?

Q3. What are the different ways supported by Spring Security for users to log into a web application?

[ The answers to these are provided in *Appendix A, Solution to Exercises.*]

Summary

In this chapter, you learned what Spring Security is and the major operations in Spring Security. We took a quick look at the Servlet filter and understood the security use case. We configured Spring Security by adding dependencies in `pom.xml` and also configured namespace

We secured the web application's URL access by providing `DelegatingFilterProxy` as the filter class and the URL pattern. We created a separated Spring Security configuration file. We saw different ways of logging into the web applicatio

We authenticated users with in-memory definition and also against the database. We saw Spring Security supports for encrypt password. Lastly, we configured the method-level security in Spring Security.

In the next chapter, we will cover Spring testing. We will understand testing using JUnit4 and TestNG. We will also understand the Mockito framework (look into MockMVC).

6

Spring Testing

In software development, testing is a crucial part. Software development cannot be completed without testing. Testing is a process that ensures the performance and quality of software development, and verifies that the applications run smoothly and flawlessly. For this, unit testing is the easiest technique. It allows us to test each component of the application separately. Integration testing ensures that multiple components are working well in a system.

To avoid the mixing of the test code with the normal code, usually unit tests are created in a separate source folder or a separate project. Some developers, on the hot topic, "What should be tested", hold that every statement in the code should be tested.

Testing can be done either automatically or manually, and automated tests can be run continuously and repeatedly at different phases of the software development process. This is highly recommended for the Agile development process. Since the Spring Framework is an Agile framework, it supports these kinds of processes.

The Java platform supports many testing frameworks, in which JUnit and TestNG are the most popular frameworks. In this chapter, we will discuss a popular Java testing framework and the basic techniques of testing. We will also discuss the support provided by the Spring Framework for unit and integrating testing.

Here is the list of topics that will be covered in this chapter:

- Testing using JUnit 4
- Testing using TestNG
- Agile software testing
- Spring MVC test framework

Testing using JUnit 4

JUnit 4 is the most widely accepted unit testing framework on the Java platform. It allows you to annotate the methods that need to be tested by using the `@Test` annotation, and it is used to create automated tests for your Java application, which can be run repeatedly to ensure the correctness of your application. The website for JUnit is <http://junit.org/>.

A `Test` class contains the JUnit tests. These are methods and are only used for testing. A test method needs to be annotated with the `@org.junit.Test` annotation. In this test method, you use a method provided by the JUnit framework to check the actual result versus the expected result of the code execution.

JUnit 4 annotations

JUnit 4 uses annotations; a few of these are listed in the following table:

Annotation	Import	Description
<code>@Test</code>	<code>org.junit.Test</code>	The <code>@Test</code> annotation identifies the test cases. A public void method annotated with the <code>@org.junit.Test</code> annotation can be run as a test case.
<code>@Before</code>	<code>org.junit.Before</code>	A public void method annotated with the <code>@Before</code> annotation is executed before each <code>Test</code> method in that class execute. It may be used to set up an environment variable.
<code>@After</code>	<code>org.junit.After</code>	A public void method annotated with the <code>@After</code> annotation is executed after each <code>Test</code> method in that class execute. It may be used to release the external resource that was allocated in a <code>Before</code> method or clean up the test environment and save memory.
<code>@BeforeClass</code>	<code>org.junit.BeforeClass</code>	A public static void method annotated with the <code>@BeforeClass</code> annotation is executed once, before all the tests in that class are executed.

Annotation	Import	Description
@AfterClass	<code>org.junit.AfterClass</code>	A public static void method annotated with the @AfterClass annotation is executed once, after all the test methods in that class have been executed. It can be used to perform some clean-up activities, such as disconnect from the database.
@Ignore	<code>org.junit.Ignore</code>	A method annotated with the @Ignore annotation will not be executed.

Assert methods

JUnit provides the static assert methods declared in the `org.junit.Assert` class to test for certain conditions. An assert method starts with `assert`, and then compares the expected value with the actual value returned by a test. The `Assert` class provides a set of assertion methods of the return type `void`. These are useful for writing tests. A few of these are listed in the table shown here:

Method	Description
<code>assertTrue(boolean expected, boolean actual)</code>	This method checks whether the Boolean condition is <code>true</code>
<code>assertFalse(boolean condition)</code>	This method checks whether the Boolean condition is <code>false</code>
<code>assertEquals(boolean expected, expected, actual)</code>	This method compares the equality of any two objects using the <code>equals()</code> method
<code>assertEquals(boolean expected, expected, actual, tolerance)</code>	This method compares either the float or the double values and tolerance defines number of the decimal that must be the same
<code>assertNull(Object object)</code>	This method tests whether a single object is <code>null</code>
<code>assertNotNull(Object object)</code>	This method tests that a single object is not <code>null</code>
<code>assertSame(Object object1, Object object2)</code>	This method tests whether two objects refer to the same object, and it must be exactly the same object pointed to
<code>assertNotSame(Object object1, Object object2)</code>	This method tests if two objects do not refer to the same object

An example of JUnit 4

Suppose we are going to develop a simple calculator. We have to test it in order to ensure the system's quality. Let's consider a simple calculator whose interface is defined as follows

```
package org.packt.Spring.chapter9.SpringTesting.Calculator;

public interface SimpleCalculator {

    public long add(int a, int b);

}
```

Now, we can implement this SimpleCalculator:

```
package org.packt.Spring.chapter9.SpringTesting.Calculator;

public class SimpleCalculatorImpl implements SimpleCalculator {

    public long add(int a, int b) {
        return a + b;
    }

}
```

Next, we will test this SimpleCalculator with JUnit 4. Most of the IDEs, such as Eclipse, STS, and NetBeans support the creation of the JUnit tests through wizards. Add JUnit 4 JAR to your CLASSPATH to compile and run the test cases created for JUnit 4, as shown here:

```
package org.packt.Spring.chapter9.SpringTesting.Calculator;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class SimpleCalculatorJUnit4Tests {

    private SimpleCalculator simpleCalculator;

    @Before
    public void init() {
        simpleCalculator = new SimpleCalculatorImpl();
    }

}
```

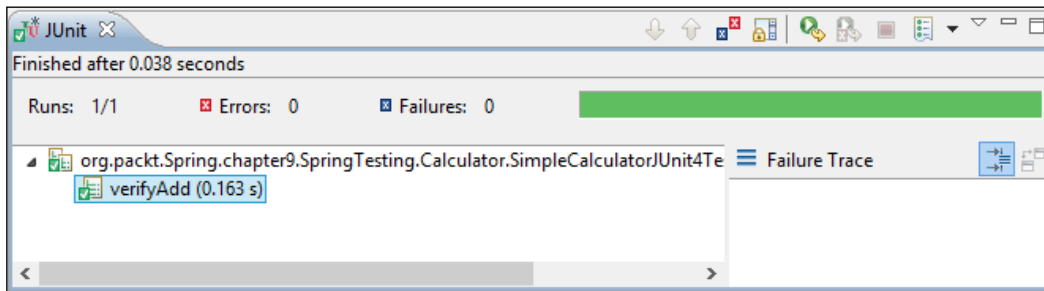
```

@Test
public void verifyAdd() {
    long sum = simpleCalculator.add(3, 7);
    assertEquals(10, sum);
}

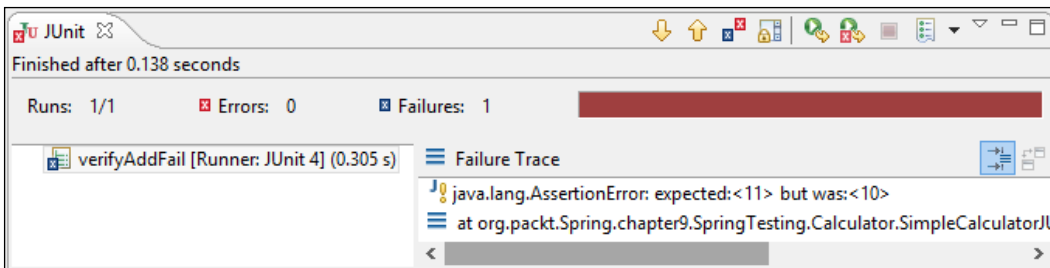
```

Now we can run our test case by right-clicking on the test, and then choosing **Run As | JUnit test**, and we can verify the JUnit view as the test case should run successfully, as shown in the following two cases:

- It will display a green bar if the test case passes:



- It will display a red bar if the test case fails:



Here is the error code in the second case:

```

@Test
public void verifyAddFail() {
    long sum = simpleCalculator.add(3, 7);
    assertEquals(11, sum);
}

```

Testing using TestNG

TestNG (Next Generation) is another testing framework that is similar to the JUnit 4 framework, but it has new functionalities such as grouping concept, and dependency testing. These have made testing easier and more powerful. It is designed to cover all the categories of tests, such as the unit test, the functional test, the integration test, and so on. TestNG also supports multi-threaded testing.

TestNG annotations

TestNG uses annotations; a few of them are listed in the following table:

Annotation	Import	Description
@Test	<code>org.testng.annotations.Test</code>	It marks the method as a test method
@BeforeMethod	<code>org.testng.annotations.BeforeMethod</code>	The annotated method will be executed before each @test annotated method
@AfterMethod	<code>org.testng.annotations.AfterMethod</code>	The annotated method will be executed after the execution of each and every @test annotated method
@BeforeClass	<code>org.testng.annotations.BeforeClass</code>	The annotated method will be executed only once before the first test method in the current class is invoked
@AfterClass	<code>org.testng.annotations.AfterClass</code>	The annotated method will be executed only once after the execution of all the @Test annotated methods of that class

Annotation	Import	Description
@BeforeTest	org.testng.annotations.BeforeTest	The annotated method will be executed before any @Test annotated method belonging to that class is executed
@AfterTest	org.testng.annotations.AfterTest	The annotated method will be executed after any @Test annotated method belonging to the classes is executed

Example of TestNG

Refer to the link <http://testng.org/doc/download.html> to set up TestNG for your IDEs. Add TestNG JAR to your CLASSPATH to compile and run the test cases created for TestNG, as shown here:

```
package org.packt.Spring.chapter9.SpringTesting.Calculator;

import org.testng.Assert;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

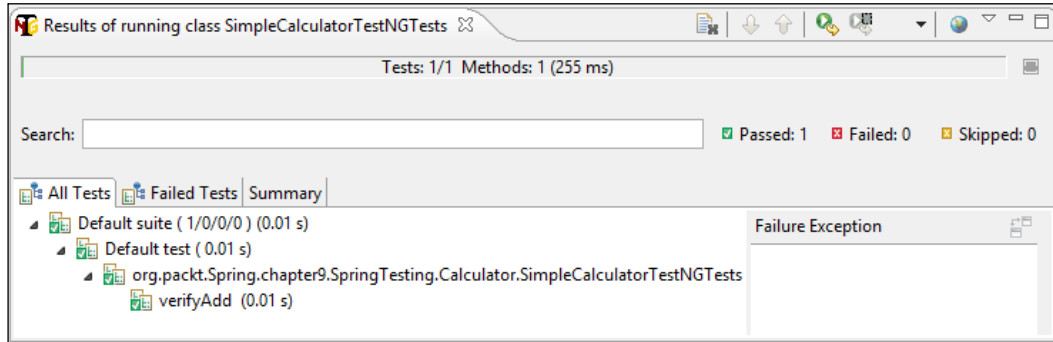
public class SimpleCalculatorTestNGTests {

    private SimpleCalculator simpleCalculator;

    @BeforeMethod
    public void beforeMethod() {
        simpleCalculator = new SimpleCalculatorImpl();
    }

    @Test
    public void verifyAdd() {
        long sum = simpleCalculator.add(3, 7);
        Assert.assertEquals(10, sum);
    }
}
```

You will see a progressive green bar if your test case passes:



Agile software testing

The term Agile, in the world of software development, typically refers to an approach to project management that aims to unite teams around the principles of collaboration, simplicity, flexibility, and responsiveness throughout the process of developing a new program in an application.

An Agile software testing means the practice of testing software for any performance issues or bugs within the context of Agile workflow. The developers and testers, in the agile approach, are seen as the two sides of the same coin. The Agile software testing includes unit testing and integration testing. It helps with executing the tests as quickly as possible.

Let's understand the significance and the objectives of unit and integration testing.

Unit testing

Unit testing, as the name suggests, is the testing of every individual method of the code. It is the method of testing the fundamental pieces of your functionality. It is a piece of code written by the software developer to test a specific functionality of the code. Unit tests are used for improving the quality of the code and preventing bugs. They are not commonly used for finding them. They are automated testing frameworks.

Let's take an example of the `EmployeeService` class that needs the `employeeDao` object for loading the data from the database. This `employeeDao` is a real object. So, to test the `EmployeeService` class, it is required to provide the `employeeDao` object that has a valid connection to the database. We also have to insert the data needed for the test into the database.

Inserting the data into the database after setting up the connection and then testing on an actual database can be a lot of work. Instead, we can provide the `EmployeeService` instance with a fake `EmployeeDao` class, which will just return the data that we need to complete the test. This fake `EmployeeDao` class will not read any data from the database. This fake `EmployeeDao` class is a mock object that is a replacement for a real object, which makes it easier to test the `EmployeeService` class.

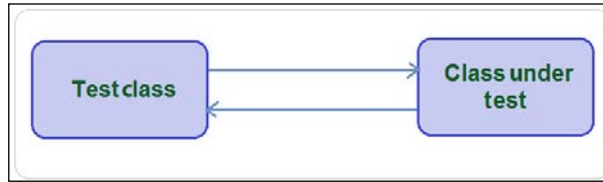
A common technique that can be applied while testing a unit that depends on other units is to simulate the unit's dependencies with stubs and mock objects, which help in reducing the complexity because of the dependencies in the unit test. Let's look at each of them in detail:

- **Stub:** A stub is a dummy object, which simulates real objects with the minimum number of methods required for a unit test. It can be configured to the return value by implementing the methods in a predetermined way along with the hardcoded data that suite the test.
- **Mock:** A mock object is a fake object or a substitute object that is added to the system, and it usually knows how its method is expected to be called for a test, and decides whether the unit test has passed or failed. The mock object tests whether the real object interacted as expected with the fake object. There may be one or more mock objects per test. A mock object is an object which mimics an actual object. In Java, there are several libraries, which are available for implementing mocking, including `jMock`, `EasyMock`, and `Mockito` (we are interested in this particular tool).

State verification is used to check whether the actual method returns the correct value. Behavior verification is used to check whether the correct method was called. Stub is used for state verification, whereas a mock object is used for behavior verification. A stub object cannot fail a unit test but a mock object can. This is because we know what and why we are implementing a stub object, whereas a mock is just a fake object that mimics a real object and if the business logic in the code is wrong, then the unit test fails even if we have passed a real object.

Unit testing for isolated classes

Unit testing is easy for the isolated class, which tests either the class or its method in isolation. Let's create unit tests for the isolated class, where the class under testing will not directly depend on any other class, as shown in the following diagram:



The core functions of the HrPayroll system should be designed around employee details. First, we need to create the Employee class and override the equals() method, as shown in the following code snippet:

```
package org.packt.Spring.chapter9.SpringTesting.modle;

public class Employee {

    private String employeeId;
    private String firstName;
    private String lastName;
    private int salary;

    // constructor, Getters and setters

    @Override
    public boolean equals(Object obj) {

        if (!(obj instanceof Employee)) {
            return false;
        }

        Employee employee = (Employee) obj;
        return employee.employeeId.equals(employeeId);
    }
}
```

Now, to persist the employee object to the HrPayroll system, we need to define the EmployeeDao interface:

```
package org.packt.Spring.chapter9.SpringTesting.dao;

import org.packt.Spring.chapter9.SpringTesting.modle.Employee;
```

```
public interface EmployeeDao {  
  
    public void createEmployee(Employee employee);  
  
    public void updateEmployee(Employee employee);  
  
    public void deleteEmployee(String employeeId);  
  
    public Employee findEmployee(String employeeId);  
  
}
```

Let's implement the `EmployeeDao` interface to demonstrate the unit testing for this isolated class:

```
package org.packt.Spring.chapter9.SpringTesting.dao;  
  
import java.util.Collections;  
import java.util.HashMap;  
import java.util.Map;  
  
import org.packt.Spring.chapter9.SpringTesting.modle.Employee;  
  
public class InMemeoryEmployeeDaoImpl implements EmployeeDao {  
  
    private Map<String, Employee> employees;  
  
    public InMemeoryEmployeeDaoImpl() {  
        employees = Collections  
            .synchronizedMap(new HashMap<String,  
Employee>());  
    }  
  
    public boolean isOldEmployee(String employeeId) {  
        return employees.containsKey(employeeId);  
    }  
  
    @Override  
    public void createEmployee(Employee employee) {  
        if (!isOldEmployee(employee.getEmployeeId())) {  
            employees.put(employee.getEmployeeId(), employee);  
        }  
    }  
}
```



```
@Override
public void updateEmployee(Employee employee) {
    if (isOldEmployee(employee.getEmployeeId())) {
        employees.put(employee.getEmployeeId(), employee);
    }
}

@Override
public void deleteEmployee(String employeeId) {
    if (isOldEmployee(employeeId)) {
        employees.remove(employeeId);
    }
}

@Override
public Employee findEmployee(String employeeId) {
    return employees.get(employeeId);
}
}
```

From the aforementioned code snippet, we can see that the `InMemoryEmployeeDaoImpl` class doesn't depend on any other class directly, which makes it easier to test, because we don't need to be worried about setting dependency and their working.

Here is an implementation of `InMemoryEmployeeDaoTest`:

```
package org.packt.Spring.chapter9.SpringTesting.test;

import junit.framework.Assert;

import org.junit.Before;
import org.junit.Test;
import org.packt.Spring.chapter9.SpringTesting.dao.
InMemoryEmployeeDao
Impl;
import org.packt.Spring.chapter9.SpringTesting.modle.Employee;

public class InMemoryEmployeeDaoTest {

    private static final String OLD_EMPLOYEE_ID = "12121";
    private static final String NEW_EMPLOYEE_ID = "53535";

    private Employee oldEmployee;
    private Employee newEmployee;
    private InMemoryEmployeeDaoImpl empDao;
```

The `setUp()` method is annotated with the `@Before` annotation, as shown in the code snippet here:

```
@Before
public void setUp() {
    oldEmployee = new Employee(OLD_EMPLOYEE_ID, "Ravi",
    "Soni", 1001);
    newEmployee = new Employee(NEW_EMPLOYEE_ID, "Shashi",
    "Soni", 3001);

    empDao = new InMemoryEmployeeDaoImpl();
    empDao.createEmployee(oldEmployee);
}
```

The `isOldEmployeeTest()` method is annotated by the `@Test` annotation. This test method verifies the `employeeId`, as shown in the following code snippet:

```
@Test
public void isOldEmployeeTest() {

    Assert.assertTrue(empDao.isOldEmployee(OLD_EMPLOYEE_ID));

    Assert.assertFalse(empDao.isOldEmployee(NEW_EMPLOYEE_ID));
}
```

The `createNewEmployeeTest()` method is annotated by the `@Test` annotation. This test method creates a new employee and then verifies the new `employeeId`:

```
@Test
public void createNewEmployeeTest() {
    empDao.createEmployee(newEmployee);

    Assert.assertTrue(empDao.isOldEmployee(NEW_EMPLOYEE_ID));
}
```

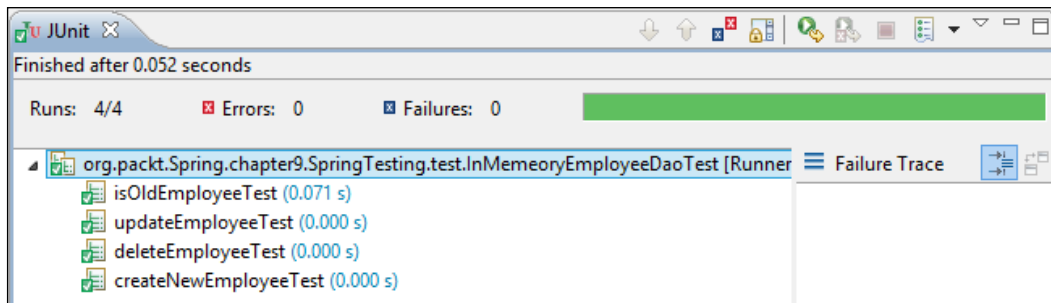
The `updateEmployeeTest()` method is annotated by the `@Test` annotation. This test method updates employee details and then verifies the employee's `firstName`, as shown here:

```
@Test
public void updateEmployeeTest() {
    String firstName = "Sharee";
    oldEmployee.setFirstName(firstName);
    empDao.updateEmployee(oldEmployee);
    Assert.assertEquals(firstName,
    empDao.findEmployee(OLD_EMPLOYEE_ID)
        .getFirstName());
}
```

The `deleteEmployeeTest()` method is annotated by the `@Test` annotation. This test method deletes employee details and then verifies the employee ID, as shown in the following code snippet:

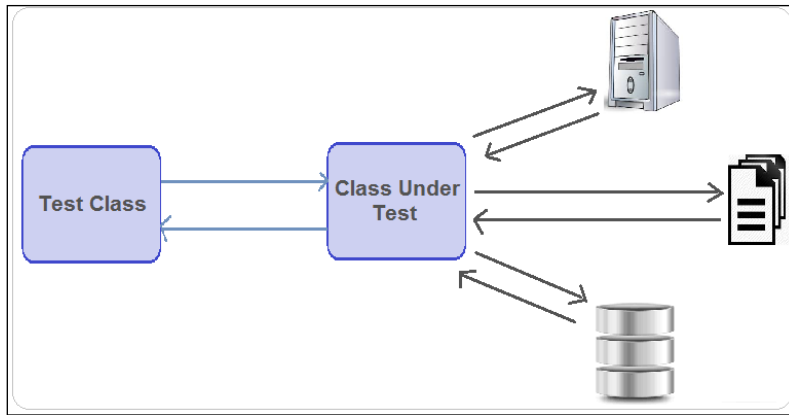
```
@Test
public void deleteEmployeeTest() {
    empDao.deleteEmployee(OLD_EMPLOYEE_ID);
    Assert.assertFalse(empDao.isOldEmployee(OLD_EMPLOYEE_ID));
}
```

The test results of the aforementioned test cases will be as shown here:

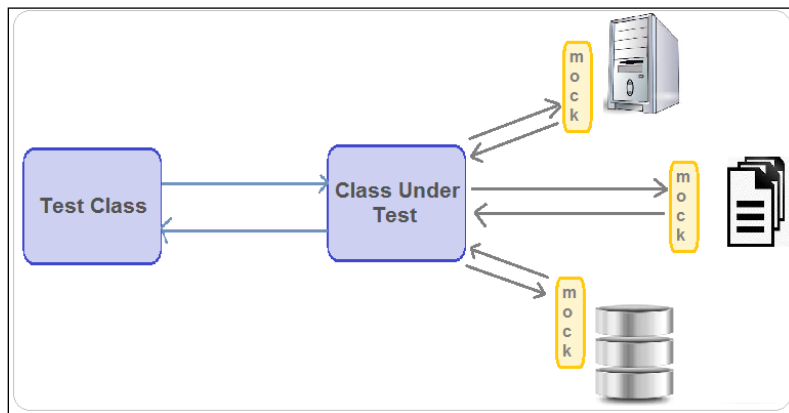


Unit testing for dependent class using mock objects

As we have seen in the previous section, testing either an isolated class or an independent class is easy. However, it would be a little more difficult to test a class that depends on another class, such as the `EmployeeService` class (that holds business logic), which depends on the `EmployeeDao` class (this class knows how to communicate with the database and get the information). Unit testing is harder and has dependencies, as shown here:



Class Under Test means that whenever we write a unit test, generally the term "unit" refers to a single class against which we have written the tests. It is the class that is being tested. So it's good to remove the dependencies, create a mock object and continue with the unit testing, as shown in the following diagram:



The concept behind removing the dependencies and creating a mock object is that by creating an object that can take the place of a real dependent object. If we are writing a unit test for our `EmployeeService` around business logic, then that particular unit test should not connect `EmployeeService` to the `EmployeeDao` intern, and then connect the `EmployeeDao` intern to the database and perform a crud operation, because we just want to perform the testing of the `EmployeeService` class, and so we need to create a mock `EmployeeDao`. The Mockito framework allows us to create the mock object.

The Mockito framework

The Mockito framework is an open source mock framework for unit testing; it was originally based on EasyMock, which can be downloaded from either <http://mockito.org/> or <https://code.google.com/p/mockito/>. It can be used in conjunction with other testing tools, such as JUnit. It helps in creating and configuring mock objects. Add the Mockito JAR to your CLASSPATH along with JUnit. It uses the field-level annotations, as shown here

- `@Mock`: This creates the mock object for an annotated field.
- `@Spy`: This creates spies for the objects or the files it annotates.
- `@InjectMocks`: The private field that is annotated by the `@InjectMocks` annotations is instantiated and Mockito injects the fields annotated with either the `@Mock` annotation or the `@Spy` annotation to it.
- `@RunWith(MockitoJUnitRunner.class)`: If you use the aforementioned annotations, then it must be done to annotate the test class with this annotation to use the `MockitoJUnitRunner`. When `MockitoJUnitRunner` executes the unit tests, it creates mock objects and spy objects for all the fields annotated by the `@Mock` annotation or the `@Spy` annotation.

Let's perform the unit testing using Mockito, where we create a mock object for a dependent object. Here is the code for the `EmployeeService.java` interface:

```
package org.packt.Spring.chapter9.SpringTesting.service;

import org.packt.Spring.chapter9.SpringTesting.modle.Employee;

public interface EmployeeService {

    public Employee findEmployee(String employeeId);

}
```

The following is an implementation of `EmployeeService`:

```
package org.packt.Spring.chapter9.SpringTesting.service;

import org.packt.Spring.chapter9.SpringTesting.dao.EmployeeDao;
import org.packt.Spring.chapter9.SpringTesting.modle.Employee;

public class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDao employeeDao = null;
```

```

    public EmployeeServiceImpl(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }

    @Override
    public Employee findEmployee(String employeeId) {
        return employeeDao.findEmployee(employeeId);
    }
}

```

And, here we have created our test class in the test folder, and created a mock object by annotating `EmployeeDao`. We have annotated the class by the `@RunWith(MockitoJUnitRunner.class)` annotation. We have created two test methods by using the `@Test` annotation, where, in the first test case, we verify that the `findEmployee` behavior happened once and in the second test case, we verify that no interactions happened on `employeeDao` mocks:

```

package org.packt.Spring.chapter9.SpringTesting.service;

import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.verifyZeroInteractions;
import static org.mockito.Mockito.when;
import junit.framework.Assert;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import org.packt.Spring.chapter9.SpringTesting.dao.EmployeeDao;
import org.packt.Spring.chapter9.SpringTesting.modle.Employee;

@RunWith(MockitoJUnitRunner.class)
public class EmployeeServiceTest {

    private static final String OLD_EMPLOYEE_ID = "12121";
    private Employee oldEmployee;
    private EmployeeService employeeService;

    @Mock
    private EmployeeDao employeeDao;

```

```
@Before
public void setUp() {
    employeeService = new EmployeeServiceImpl(employeeDao);
    oldEmployee = new Employee(OLD_EMPLOYEE_ID, "Ravi",
"Soni", 1001);
}

@Test
public void findEmployeeTest() {

    when(employeeDao.findEmployee(OLD_EMPLOYEE_ID)).
thenReturn(oldEmployee);
    Employee employee =
employeeService.findEmployee(OLD_EMPLOYEE_ID);
    Assert.assertEquals(oldEmployee, employee);

    // Verifies findEmployee behavior happened once
    verify(employeeDao).findEmployee(OLD_EMPLOYEE_ID);

    // asserts that during the test, there are no other
calls to the mock
    // object.
    verifyNoMoreInteractions(employeeDao);
}

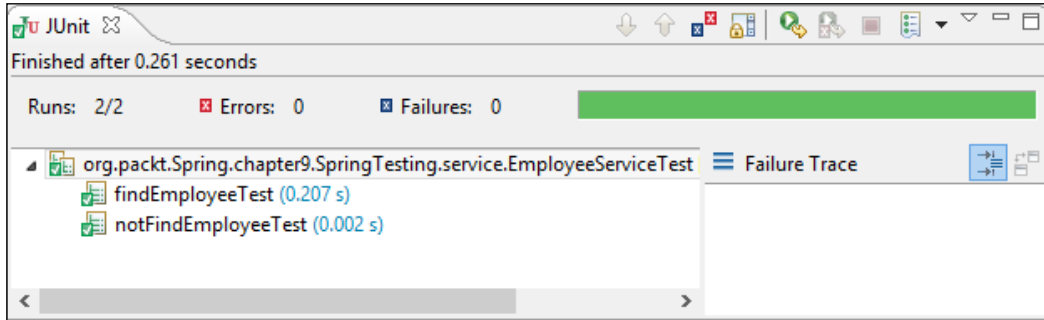
@Test
public void notFindEmployeeTest() {

    when(employeeDao.findEmployee(OLD_EMPLOYEE_ID)).thenReturn
(null);
    Employee employee =
employeeService.findEmployee(OLD_EMPLOYEE_ID);
    Assert.assertNotSame(oldEmployee, employee);

    verify(employeeDao).findEmployee(OLD_EMPLOYEE_ID);

    // Verifies that no interactions happened on employeeDao
mocks
    verifyZeroInteractions(employeeDao);
    verifyNoMoreInteractions(employeeDao);
}
}
```

And, the result of running the test as JUnit is as follows:



Integration testing

Integration testing is a phase of software testing in which individual software modules are combined and tested as a group to ensure that the required units are properly integrated and interact correctly with each other. The purpose of integration testing is to verify the functionality, performance, and reliability of the code. Integration testing is used for testing several units together.

Let's take an example. We can create an integration test to test `EmployeeServiceImpl` using `InMemoryEmployeeDaoImpl` as a DAO implementation:

```
package org.packt.Spring.chapter9.SpringTesting.service;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.packt.Spring.chapter9.SpringTesting.dao.EmployeeDao;
import org.packt.Spring.chapter9.SpringTesting.dao.
InMemoryEmployeeDaoImpl;
import org.packt.Spring.chapter9.SpringTesting.modle.Employee;

public class EmployeeServiceIntegrationTest {

    private static final String OLD_EMPLOYEE_ID = "12121";
    private static final String NEW_EMPLOYEE_ID = "53535";

    private Employee oldEmployee;
    private Employee newEmployee;
    private EmployeeService employeeService;
```



```
@Before
public void setUp() {
    oldEmployee = new Employee(OLD_EMPLOYEE_ID, "Ravi",
    "Soni", 1001);
    newEmployee = new Employee(NEW_EMPLOYEE_ID, "Shashi",
    "Soni", 3001);

    employeeService = new EmployeeServiceImpl(
        new InMemoryEmployeeDaoImpl());
    employeeService.createEmployee(oldEmployee);
}

@Test
public void isOldEmployeeTest() {

    Assert.assertTrue(employeeService.isOldEmployee
    (OLD_EMPLOYEE_ID));

    Assert.assertFalse(employeeService.isOldEmployee
    (NEW_EMPLOYEE_ID));
}

@Test
public void createNewEmployeeTest() {
    employeeService.createEmployee(newEmployee);

    Assert.assertTrue(employeeService.isOldEmployee
    (NEW_EMPLOYEE_ID));
}

@Test
public void updateEmployeeTest() {
    String firstName = "Sharee";
    oldEmployee.setFirstName(firstName);
    employeeService.updateEmployee(oldEmployee);
    Assert.assertEquals(firstName,

    employeeService.findEmployee(OLD_EMPLOYEE_ID).getFirstName());
}

@Test
public void deleteEmployeeTest() {
    employeeService.deleteEmployee(OLD_EMPLOYEE_ID);
```

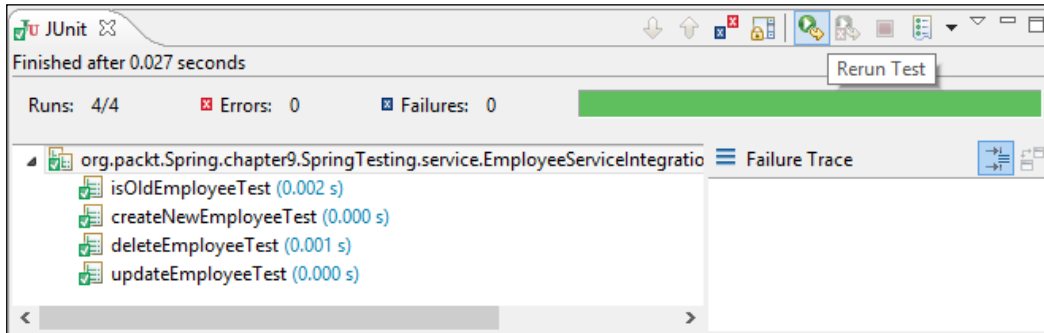
```

        Assert.assertFalse(employeeService.isOldEmployee
(OLD_EMPLOYEE_ID));
    }

}

```

The result is shown here:



Create unit tests of the Spring MVC controller

We will take the example of the Spring MVC from this chapter as a target application to test and execute unit testing. We have the `EmployeeController` class as a target class to test.

You'll find the following code in `EmployeeController.java`:

```

package org.packt.Spring.chapter7.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/employee")
public class EmployeeController {

    @RequestMapping(method = RequestMethod.GET)
    public String welcomeEmployee(ModelMap model) {

```

```
        model.addAttribute("name", "Hello World!");
        model.addAttribute("greetings",
                           "Welcome to Packt Publishing - Spring MVC
!!!");
        return "hello";
    }
}
```

In the aforementioned code snippet, the `welcomeEmployee()` method in the `EmployeeController` class gets mapped into the HTTP request. In the `welcomeEmployee()` method, the request is processed and bound to the model objects. Then, the `EmployeeController` class updates the model and the view state, and after this it returns to the logical view.

The main objective of the unit testing controller class is to verify that the methods of the controller class update the model and the view states properly and also return to the correct view. Since we perform the testing of the controller class's behavior, we should mock the service layer (if present) with the correct behavior.

For the `EmployeeController` class, we would like to develop the test cases for the `welcomeEmployee()` method. Here we will test the `welcomeEmployee()` method of the controller using JUnit 4.

It is important to note that the classes undergoing testing should be placed in the folder `/src/test/java` and the resources filed should be placed in the folder `/src/test/resources`.

You'll find this code in `EmployeeControllerTest.java`:

```
package org.packt.Spring.chapter7.springmvc.controller;

import org.junit.Assert;
import org.junit.Test;
import org.packt.Spring.chapter7.springmvc.controller.
EmployeeController;
import org.springframework.ui.ExtendedModelMap;
import org.springframework.ui.ModelMap;

public class EmployeeControllerTest {

    @Test
    public void test () {

        EmployeeController controller = new EmployeeController();
```

```

    ModelMap modelMap = new ExtendedModelMap();

    String view = controller.welcomeEmployee(modelMap);

    // verify view page name
    Assert.assertNotNull(view);
    Assert.assertEquals("hello", view);

    // verify page title
    String titlename = modelMap.get("name").toString();
    Assert.assertEquals("Hello World!", titlename);

    // verify greeting message
    String greetings = modelMap.get("greetings").toString();
    Assert.assertEquals("Welcome to Packt Publishing - Spring
MVC !!!",
                       greetings);
}
}

```

Even though the preceding code works, it has the following problems:

- The preceding test case tests the controller API strictly but disagrees on the request methods, such as GET, POST, PUT, or DELETE
- The preceding test case only tests the return value that is put in the ModelMap
- The preceding test case tests for the correct view name

It is always challenging to perform unit testing of web applications. A better solution for the aforementioned problem is provided by the Spring MVC test framework, which allows us to test the Spring MVC controller.

Spring MVC test framework

The Spring MVC test framework makes unit testing and integration testing of the Spring MVC controller more meaningful by offering first class JUnit support. It helps in testing all the aspects of the controller method that have not been tested before. It allows us to test these aspects in depth without starting a web container.

In order to perform a test on the Spring MVC framework, the Spring TestContext framework along with JUnit or TestNG makes it so simple by providing an annotation-driven unit and integration testing support. The Spring TestContext framework can be tested by annotations such as, @RunWith, @WebAppConfiguration, and @ContextConfiguration, to load the Spring configuration and inject the WebApplicationContext into the MockMvc for the unit and the integration test.

Required dependencies

We can configure the Spring `TestContext` framework by updating `pom.xml` with the required dependencies, such as `spring-test`, `junit`, and `mockito-all`. The following table explains them in detail:

Group ID	Artifact ID	Version	Description
org.springframework	spring-test	3.2.4 release	It supports unit and integration testing of the Spring components
org.mockito	mockito-all	1.9.5	The library of the Mockito mocking framework
JUnit	junit	4.10	The library of the JUnit framework

You'll find the following code at `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.packt.Spring.chapter7.springmvc</groupId>
  <artifactId>SpringMVCPayrollSystem</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringMVCPayrollSystem Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <properties>
    <spring.version>3.2.0.RELEASE</spring.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring.version}</version>
```

```

</dependency>
<!-- Servlet -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.5</version>
  <scope>provided</scope>
</dependency>
<!-- Test -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>3.2.4.RELEASE</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-all</artifactId>
  <version>1.9.5</version>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.10</version>
  <scope>test</scope>
</dependency>
</dependencies>
<build>
  <finalName>SpringMVCPayrollSystem</finalName>
</build>
</project>

```

Annotations in Spring testing

The Spring Framework provides the annotations that can be used to perform unit and integration testing with the `TestContext` framework. Here, we will discuss the two important annotations: `@ContextConfiguration` and `@WebAppConfiguration`.

The @ContextConfiguration annotation

This annotation is used to set `ApplicationContext` for the test classes by taking the actual configuration file with the file path. In the following code, we have given a file, so it will take the relative path as the root package. We can also give the exact path by specifying the file: prefix. Also, we can pass more than one configuration file using a comma separator, as shown here

```
@ContextConfiguration ({ "classpath*: SpringDispatcher-  
servlet.xml" })  
public class EmployeeControllerTestWithMockMvc {  
    // class body  
}
```

The `@ContextConfiguration` annotation caches the `ApplicationContext` for us and puts it in the static memory for the entire duration of the test or the test suite. And the entire test executes it in the same JVM because `ApplicationContext` is stored in the static memory. If the second JVM is there, it will not have access to the static context, and it will result in a second `ApplicationContext` being created.

The @WebAppConfiguration annotation

It is a class-level annotation used to create a web version of the application context in the Spring Framework. It is used to denote that the `ApplicationContext`, which is loaded for an integration test and used by that class, is an instance of `WebApplicationContext`. It is important to note that the `@WebAppConfiguration` annotation must be used with the `@ContextConfiguration` annotation:

```
@WebAppConfiguration  
@ContextConfiguration ({ "classpath*: SpringDispatcher-  
servlet.xml" })  
public class EmployeeControllerTestWithMockMvc {  
    // class body  
}
```

MockMvc

The `MockMvc` is a key part of the Spring MVC Test framework, which can be used to write the tests for the applications developed using the Spring MVC. It is the entry point for Spring MVC testing. The `MockMvc` mock the entire Spring MVC infrastructure and is created using the implementations of the `MockMvcBuilder` interface. In order to use the Spring MVC testing, the first step is to create an instance of `MockMvc`. There are four static methods in the `MockMvcBuilders` class.

They are as follows:

- `ContextMockMvcBuilder annotationConfigSetup(Class... configClasses)`: Use this method when you need to configure the application context using Java configuration.
- `ContextMockMvcBuilder xmlConfigSetup(String... configLocations)`: Use this method when you need to configure the application context by using the XML configuration files.
- `StandaloneMockMvcBuilder standaloneSetup(Object... controllers)`: You can use this method when you need to configure the test controller manually, and when you want to run the individual components for testing. We don't need to configure the entire application context; instead we only need to configure and execute the associated controller component files.
- `InitializedContextMockMvcBuilder webApplicationContextSetup(WebApplicationContext context)`: This method must be used when you have already fully initialized the `WebApplicationContext` object.

Here, we have created the `MockMvc` instance using `MockMvcBuilders` and calling the `standaloneSetup()` method after passing an instance of the controller class as a parameter and then building it by calling the `build()` method, as shown in the following code snippet:

```
private MockMvc mockMvc;

@Before
public void setup() {
    this.mockMvc = MockMvcBuilders.standaloneSetup
        (employeeController).build();
}
```

Once we have an instance of `MockMvc`, we can perform the testing using `MockMvc`. We can send the HTTP request after specifying all the details, such as the HTTP method, the content type, and so on. And then, we can verify the results.

Assertion

To perform the assertion, first we use the instance of `MockMvc` and then we call the `perform()` method to pass a relative path to run the test case. And then, we can verify the different components inside the controller using `andExpect`. The `andExpect(status().isOk())` is used to check for a 200 status. Similarly, we can perform the `contentType` validation, the `xpath` validation, validate data in the model, URL validation, and the view name validation.

The sample code for this is as shown here:

```
this.mockMvc
    .perform(get("/employee"))
    .andExpect(status().isOk())
    .andExpect(view().name("hello"))
    .andExpect(model().attribute("name", "Hello
World!"))
    .andExpect(
        model().attribute("greetings",
            "Welcome to Packt
Publishing - Spring MVC !!!"));
```

@RunWith(SpringJUnit4ClassRunner.class)

This is a JUnit annotation. It executes the tests in a class annotated by the `@RunWith` annotation, or extends a class annotated by the `@RunWith` annotation by invoking the class passed as a parameter, which means that the tests in the annotated class are not executed by the in-built API in the JUnit framework, the runner class used to execute the test case. In order to use the Spring's JUnit class runner for running the test cases within the Spring's `ApplicationContext` environment, passed spring's `SpringJUnit4ClassRunner` class as parameter.

So now, we have the complete code to perform the testing of the `EmployeeController` controller using the Spring MVC test framework. We will use the `MockMvc` that will mock the entire Spring MVC infrastructure. We will create a `MockMvc` instance in the method annotated by the `@Before` annotation, so that it will be available before the test starts.

You'll find this code in `EmployeeControllerTestWithMockMvc.java`:

```
package org.packt.Spring.chapter7.springmvc.controller;

import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilder
s.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.
status;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.
view;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.
model;
```

```

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.MockitoAnnotations;
import org.packt.Spring.chapter7.springmvc.controller.
EmployeeController;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration({ "classpath*:SpringDispatcher-servlet.xml"
})
public class EmployeeControllerTestWithMockMvc {

    @InjectMocks
    private EmployeeController employeeController;

    private MockMvc mockMvc;

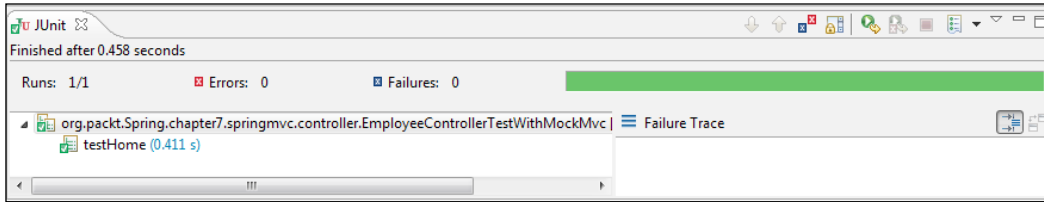
    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
        this.mockMvc =
MockMvcBuilders.standaloneSetup(employeeController).build();
    }

    @Test
    public void testHome() throws Exception {

        this.mockMvc
            .perform(get("/employee"))
            .andExpect(status().isOk())
            .andExpect(view().name("hello"))
            .andExpect(model().attribute("name", "Hello
World!"))
            .andExpect(
                model().attribute("greetings",
                    "Welcome to Packt
Publishing - Spring MVC !!!"));
    }
}

```

Now, we can run the test case by right-clicking on the test and then choosing **Run As | JUnit Test**. We can verify in the JUnit view as the test case should run successfully, as shown here:



Exercise

- Q1. What is the difference between JUnit 4 and TestNG?
- Q2. What is the difference between unit testing and integration testing?
- Q3. Explain the Spring MVC test framework.
- Q4. Explain `@ContextConfiguration` and `@WebAppConfiguration`.
- Q5. Explain `MockMvc` and `@RunWith(SpringJUnit4ClassRunner.class)`.



The answers to these are provided in *Appendix A, Solution to Exercises*.

Summary

In this chapter, you learned about the Spring test. We understood testing using JUnit 4, its annotations, its assert statements, and demonstrated all this with an example. Then, we moved on to testing using TestNG and its annotations. We also demonstrated this with an example. We understood Agile software testing, which includes unit testing and integration testing. And then, we went through unit testing for the isolated classes and then we went through the Mockito framework for the dependent class. We also looked into integration testing with the help of an example. Then, we created a unit test for the Spring MVC controller using JUnit. And finally, we discussed the topic of the Spring MVC test framework, where we saw the dependencies required to use the Spring MVC test and the annotations provided by them. And then, we looked into `MockMvc` and their assertion method.

In the next chapter, we will go through the e-mail support in Spring to develop an e-mail application. We will then look at the JavaMail API and the Spring API to write e-mails. You will also learn to develop a simple Spring e-mail application.

7

Integrating JavaMail and JMS with Spring

In this chapter, first, we will go through the e-mail support in Spring to develop an e-mail application. We will then look into the JavaMail API and the Spring API for e-mails. Later in this chapter, you will learn to develop a simple Spring e-mail application.

In an e-mail application, an e-mail composed by a client is sent to a server and then delivered to the destination and then sends back a response to the client. Here, the communication between the client and the server is completely synchronous, which can be enriched by making this communication asynchronous. The Java messaging system is the standard **application programming interface (API)** to perform asynchronous communication.

Secondly, we will cover JMS and what message and messaging is. Then, we will look into the JMS application and its components. We will also cover MOM Service Provider and the configuration of ActiveMQ as Message Queue. Then, we will configure a Spring bean in the Spring configuration file, and using Spring J template, we will create a `MessageSender` class and run an application to perform a functionality related to JMS.

The list of the topics covered in this chapter is as follows:

- E-mail support in Spring
- Spring Java Messaging Service

E-mail support in Spring

Electronic mail (e-mail) plays an important role in all day-to-day activities in this era of global networks. Suppose you want to get periodic updates of a particular feature on a website, by just subscribing to that feature, you will start receiving e-mails regarding these updates. E-mails also allow you to send notifications, business orders, or any periodic reports of a producer.

Oracle provides a simple yet powerful API known as a JavaMail API for creating an application with an e-mail support. The JavaMail API provides a set of classes and interfaces for creating an e-mail application. This API is used to programmatically send and receive e-mails, which can be scaled up to work with different protocols associated with the mailing system. Although it is a powerful API, it is very complex, and using the JavaMail API directly in our application is a slightly tedious task as it involves writing a lot of code.

The Spring Framework provides a simplified API and plugging for a full e-mail support, which minimizes the effect of the underlying mailing system specifications. The Spring e-mail support provides an abstract, easy, and implementation-independent API for sending e-mails. In this chapter, we will get an overview on the JavaMail API and learn how to send e-mail using the JavaMail API in Spring.

Introducing the JavaMail API

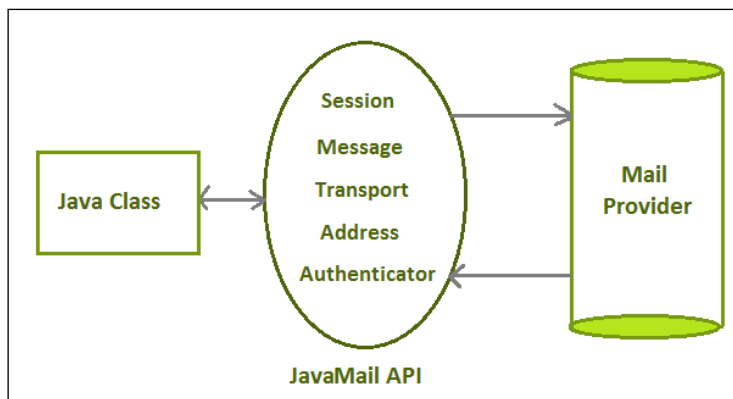
The JavaMail API provides a protocol-independent and platform-independent framework to provide e-mail support for a Java application. The JavaMail API is a collection of classes and interfaces that comprise an e-mail system. The steps involved in sending a simple e-mail using the JavaMail API are as follows:

1. Connect to an e-mail server by specifying the username and password; let's say for example, if you want to send an e-mail from `abc@xyz.com`, then you need to connect to the e-mail server of `xyz.com`.
2. Create a message by specifying the recipient's addresses that can include Cc and Bcc addresses as well.
3. Add attachments to the message if any.
4. Transport the message to the e-mail server.

Sending a simple e-mail requires the use of a number of classes and interfaces that are present in the `javax.mail` and `javax.mail.internet` packages. The important classes and interfaces in the JavaMail API are listed in the following table:

Class/interface	Description
<code>Session</code>	The <code>javax.mail.Session</code> is the key class of the JavaMail API. It represents an e-mail session. Typically, we create a <code>Session</code> object, set the properties, and send a message.
<code>Message</code>	The <code>javax.mail.Message</code> is an abstract class of the JavaMail API that models an e-mail message. It represents the e-mail sent.
<code>Transport</code>	The <code>javax.mail.Transport</code> is an abstract class of the JavaMail API that represents the protocol used to send and receive e-mails. The <code>Transport</code> object is used for sending an e-mail message.
<code>Authenticator</code>	The <code>javax.mail.Authenticator</code> is an abstract class of the JavaMail API that represents an authentication for the e-mail provider.
<code>PasswordAuthentication</code>	The <code>PasswordAuthentication</code> holds the username and password by the <code>Authenticator</code> object.
<code>MimeMessage</code>	The <code>javax.mail.internet.MimeMessage</code> is an abstract class that represents a Multipurpose Internet Mail Extension (MIME) message. <code>MimeMessage</code> is an e-mail message, which will understand the MIME types and headers.
<code>InternetAddress</code>	The <code>javax.mail.internet.InternetAddress</code> represents an Internet e-mail address such as To, Bcc, and Cc.

The JavaMail application uses the JavaMail API to exchange e-mails, as shown in the following figure



Using the JavaMail API

The JavaMail API can be used to create a class to send an e-mail using the `MailHelper` class. This `MailHelper` class contains a constructor, which can be used to initialize the host, username, and password. This `MailHelper` class also contains the `sendMail()` method.

The following code snippet shows the `MailHelper.java` class:

```
public class MailHelper {

    private Properties props;
    private String host;
    private String userName;
    private String password;

    public MailHelper(String host, String username, String
password) {
        this.userName = username;
        this.password = password;
        props = new Properties();
        // put host information
        props.put("mail.smtp.host", host);
        // put true for authentication mechanism
    }
}
```

```

        props.put("mail.smtp.auth", "true");
    }

    public void sendMail(String from, String to, String subject,
String body){
        Session session = Session.getDefaultInstance(props, new
PasswordAuthenticator());
        try{
            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress(from));
            InternetAddress toAddress = new
InternetAddress(to);
            message.addRecipient(RecipientType.TO, toAddress);
            message.setSub(subject);
            message.setText(body);
            Transport transport =
session.getTransport("smtp");
            transport.connect();
            transport.sendMessage(message,
message.getAllRecipients());
            transport.close();
        } catch(NoSuchProviderException ex){
            ex.printStackTrace();
        } catch(MessagingException ex){
            ex.printStackTrace();
        }
    }

    private class PasswordAuthenticator extend Authenticator{
        protected PasswordAuthentication
getPasswordAuthentication(){
            return new PasswordAuthentication(userName,
password);
        }
    }
}

```

In the preceding code snippet, the `MailHelper` class has instance variables and a constructor. The `props` variable, of the `Properties` collection type, used to specify the common properties to connect to the e-mail provider host.

Simple Mail Transfer Protocol (SMTP) is a protocol that is used to send e-mails. The SMTP server performs this job. The `props.put("mail.smtp.host", host)` code inside the constructor is used to specify the host information, where we connect to the SMTP server of our host. The `props.put("mail.smtp.auth", "true")` code inside the constructor specifies the use of an authentication mechanism to connect to the SMTP server.

The `sendMail` method of the `MailHelper` class creates a `Session` object using the host information and the username-password credentials. The from-address and the to-address are added to the instance of the `MimeMessage` class. The subject and the body of the e-mail are also added to this `MimeMessage` object. Finally, the message is sent using an instance of the `Transport` class.

The `PasswordAuthenticator` class is an inner class that has been used by a `Session` object to hold the username and the password.

From the preceding code, the following problems can be encountered:

- Lots of initialization and creation work involved in sending a simple e-mail
- Exceptions need to be taken care of while using the JavaMail API; this results in some extra lines of code
- Some extra classes are needed if required to perform the attachment operation while sending an e-mail using the JavaMail API
- A solution to the preceding problem is provided by the Spring Framework that simplifies the use of the JavaMail API to send e-mails

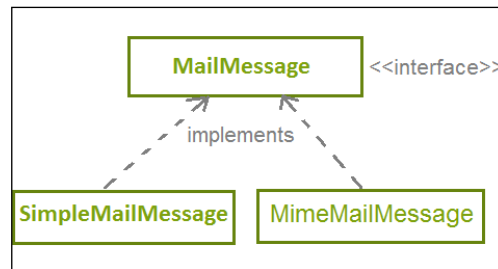
Let's understand the use of the Spring API for the JavaMail API and rewrite the `MailHelper` class.

The Spring API for JavaMail

The Spring Framework provides an API to simplify the use of the JavaMail API. The classes handle the initialization, clean-up operations, and exceptions. The packages for the JavaMail API provided by the Spring Framework are listed in the following table:

Package	Description
<code>org.springframework.mail</code>	This defines the basic set of classes and interfaces for sending e-mails
<code>org.springframework.mail.java</code>	This defines JavaMail API-specific classes and interfaces for sending e-mails

In the Spring mail API hierarchy, the `org.springframework.mail` package is the root-level package for the Spring Framework's e-mail support, as shown here:



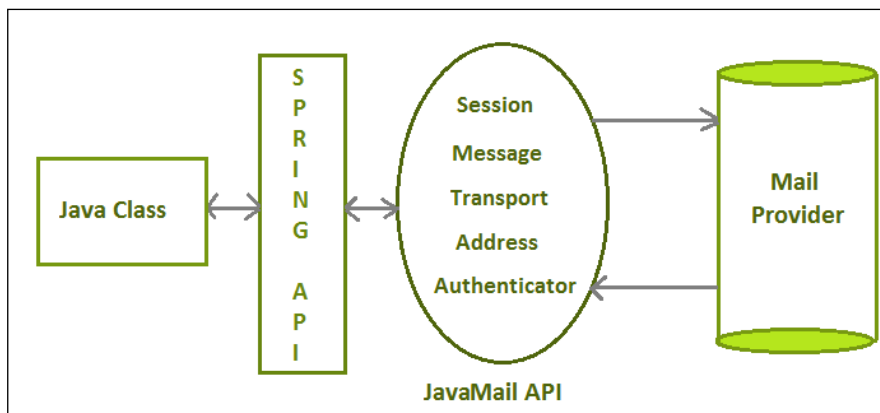
The important classes and interfaces in the `org.springframework.mail` package are listed in the following table:

Class/interface	Description
MailMessage	Refers to a common interface for all types of messages that can be sent. It doesn't support complex MIME messages. It is used for sending simple plain-text e-mails.
MailSender	Refers to an interface that defines methods for sending simple e-mails. It supports only the plain-text e-mails.
MailException	Refers to a base class for all the exceptions thrown by the mailing system.
SimpleMailMessage	Refers to the class that defines the representation of a simple message that can be sent.

The important classes and interfaces in the `org.springframework.mail.java` package are listed in the following table:

Class/interface	Description
JavaMailSenderImpl	Refers to the implementation of the JavaMailSender interface. It is a core class that is used to send simple as well as MIME messages. It extends the MailSender interface and provides the methods for constructing and sending a MIME message.
MimeMailMessage	Implements the MailMessage interface. It is based on the JavaMail MIME message.
MimeMessageHelper	Acts as a wrapper for a MIME message. It is used to populate a MIME message and is used by the JavaMailSenderImpl class.

The Java application can use Spring to access the JavaMail API for sending e-mails, as shown in the following figure



In the preceding figure, the Java classes use the Spring API, which indirectly uses the JavaMail API to send e-mails.

Developing a Spring Mail Application

Let's create a JavaMail API with the Spring application for sending e-mails via the Gmail SMTP server using the Spring mail API. Here, we develop a basic e-mail application that creates simple e-mails containing text only.

Configuration file – Spring.xml

Let's now create the configuration file `Spring.xml`, and configure the `mailSender` bean of the `JavaMailSenderImpl` class and define its properties

- `host`
- `port`
- `username`
- `password`

Also, configure the bean for the `EmailService` class with the `mailSender` property:

```
<!-- SET default mail properties -->
<bean id="mailSender" class="org.springframework.mail.javamail.
JavaMailSenderImpl">
    <property name="host" value="smtp.gmail.com" />
    <property name="port" value="25" />
</bean>
```

```

<property name="username" value="username" />
<property name="password" value="password" />

<property name="javaMailProperties">
    <props>
        <prop key="mail.smtp.auth">true</prop>
        <prop key="mail.smtp.starttls.enable">true</prop>
    </props>
</property>
</bean>

<bean id="emailService" class="org.packt.Spring.chapter10.mail">
    <property name="mailSender" ref="mailSender" />
</bean>

```

The preceding configuration file sets the host as "smtp.gmail.com" and the port as "25." The username and the password properties need to be set with reader's username and password of their Gmail account. The username is used as the sender of the e-mail.

Spring's e-mail sender

It is the e-mail API-specific Java file. It provides the definition of the `sendEmail()` method, which is used to send the actual e-mail to the recipient:

```

package org.packt.Spring.chapter10.mail;

import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class EmailService
{
    @Autowired
    private MailSender mailSender;

    public void sendEmail(String to, String subject, String msg) {

        // creates a simple e-mail object
        SimpleMailMessage email = new SimpleMailMessage();

        email.setTo(to);
        email.setSubject(subject);
        email.setText(msg);
    }
}

```

```
        // sends the e-mail
        mailSender.send(email);
    }
}
```

Here, we have autowired MailSender and called the send() method that will send the e-mails.

The MailerTest class

The MailerTest class has the main() method that will call the sendEmail() method of the EmailService class and send an e-mail:

```
package org.packt.Spring.chapter10.mail;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
ApplicationContext;

public class MailerTest
{
    public static void main( String[] args )
    {
        //Create the application context
        ApplicationContext context =
            new ClassPathXmlApplicationContext("Spring.xml");

        //Get the mailer instance
        EmailService emailService =
            (EmailService)context.getBean("emailService ");

        //Send a composed mail
        emailService.sendEmail("****@gmail.com",
            "Email Test Subject",
            "Email Testing body");
    }
}
```

The output of this application can be confirmed by opening the inbox

We have developed an application using Spring e-mails. Let's now understand the Spring Java Messaging Service.

Spring Java Messaging Service

In this section, first, we will go through the basics of Java Messaging Service and will see the differences between JMS and e-mail. Then, we will look into the JMS application and its different components that create the complete JMS application. We will also look through other things such as the JMS provider and the messaging model. We will dig into the API programming model. Then, we will see the messaging consumption types. Then, we will jump into the Spring JMS integration and we will see some code samples. And then, we will look into the details of the code content.

Let's now discuss the message and messaging.

What is a message and messaging?

A message is nothing but just bytes of data or information exchanged between two parties. By taking different specifications, a message can be described in various ways. However, it is nothing but an entity of communication. A message can be used to transfer a piece of information from one application to another application, which may or may not run in the same platform.

Messaging is the communication between different applications (in a distributed environment) or system components, which are loosely coupled unlike its peers such as TCP sockets, **Remote Method Invocation (RMI)**, or CORBA, which is tightly coupled. The advantage of Java messaging includes the ability to integrate different platforms, increase the scalability and reliability of message delivery, and reduce the system bottlenecks. Using messaging, we can increase the systems and clients who are consuming and producing the message as much as we want.

We have quite a lot of ways in which we communicate right from the instance messenger, to the stock taker, to the mobile-based messaging, to the age-old messaging system; they are all part of messaging. We understand that a message is a piece of data transferred from one system to another and it can be between humans as well, but it is mainly between systems rather than human beings when we talk about the messaging using JMS.

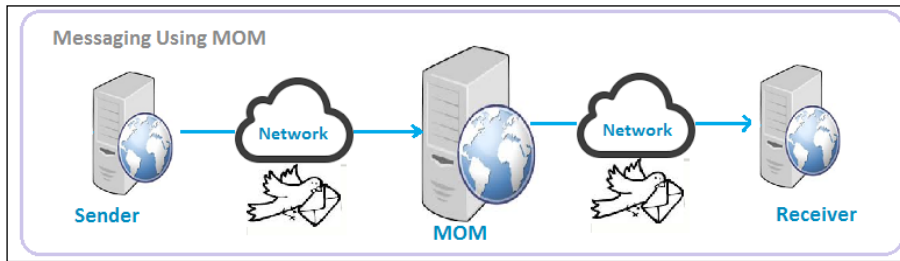
What is JMS?

The **Java Message Service (JMS)** is a **Java Message Oriented Middleware (MOM)** API for sending messages between two or more clients. JMS is a part of the Java Enterprise edition. JMS is a broker like a postman who acts like a mediator between the message sender and receiver.

JMS is a specification that describes a common way for Java programs to create, send, and read distributed enterprise messages. It advocates the loosely coupled communication without caring about the sender and the receiver. It provides asynchronous messaging, which means that it doesn't matter whether the sender and the receiver are present at the same time or not. The two systems that are sending or receiving messages need not be up at the same time.

The JMS application

Let's look into the sample JMS application pictorial as shown in the following figure



We have a **Sender** and a **Receiver**. The **Sender** sends a message while the **Receiver** receives one. We need a broker that is **MOM** between the **Sender** and the **Receiver** who takes the sender's message and passes it to the network to the receiver. **MOM** is basically an MQ application such as ActiveMQ or IBM-MQ, which are two different message providers. The **Sender** promises the loose coupling and it can be a .NET or mainframe-based application. The **Receiver** can be a Java or Spring-based application, and it sends back the message to the **Sender** as well. This is a two-way communication that is loosely coupled.

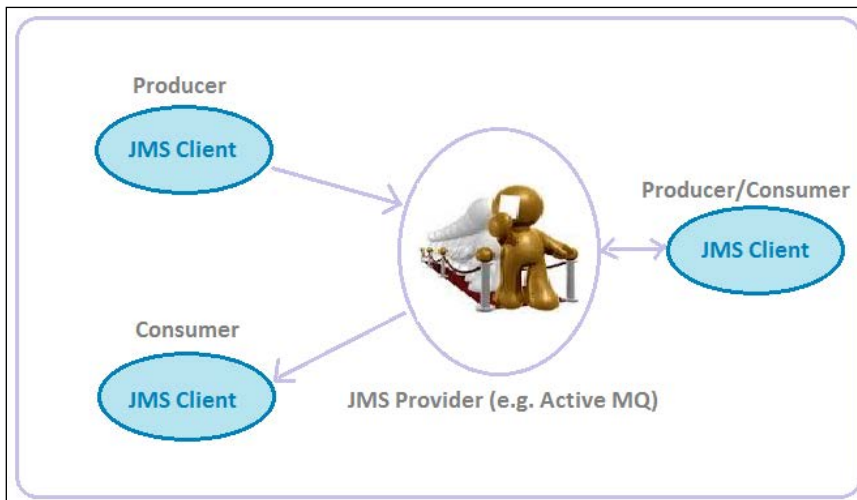
JMS components

Let's move on the JMS components listed in the following table:

Component	Description
JMS provider	<p>The JMS provider is the messaging system (that is, MOM) and acts as a message broker or agent as like a post office or postman. It implements JMS in addition to other administrative and control functionalities required of a full-featured messaging product (Active MQ or IBM MQ).</p> <p>It is an agent or message broker that takes the messages and sends them across. It is like a post office or postman that takes your e-mail and delivers it to the recipient.</p>

Component	Description
JMS client	The JMS client is a Java application that receives or produces messages. The JMS client is a Java application. It is the one who is producing or receiving the messages. Let's say that you are sending a postcard to your friend; then, you and your friend are the JMS client.
JMS producer/publisher	The JMS producer and publisher are two types of JMS client that creates and sends messages.
JMS consumer/subscriber	The JMS consumer and subscriber are two types of JMS clients that receive messages.
JMS application	The JMS application is the system composed of typically one JMS provider and many JMS clients.

Here is the pictorial representation:



There are three JMS clients in the preceding figure. The **Producer** can be assumed as it's you who is going to send a message to your friend. The **Consumer** can be assumed to be your friend who will receive a message. The **Producer/Consumer** could be someone else who will receive as well as send a message. The **JMS Provider** can be assumed as the post office or postman via which the whole delivery things happen and which guarantee that the sure delivery happens only once.

MOM Service Provider

There are various MOM Service Provider products; some of them are listed in the following table:

Product	Company
WebLogic	Oracle
MQ Series	IBM
JBOSSMQ	JBOSS
SoniqMQ	Progress
ActiveMQ	Apache

We will mainly look into the ActiveMQ message queue. The Active MQ is from Apache, and it's free.

Configuring ActiveMQ – message queue

We need to follow the given steps to configure ActiveMQ to our system

1. While configuring ActiveMQ to our system, we need to download the ZIP distribution from the official link <http://activemq.apache.org/download.html>, as shown in the following screenshot:



2. Then, extract the ZIP distribution to a folder.
3. Navigate to the `activemq-5.10.0\bin` folder, inside which you will find the following folders:
 - `activemq-5.10.0\bin` for 64 bit
 - `activemq-5.10.0\bin` for 32 bit

These folders can be seen in the following screenshot:

Name	Date modified	Type	Size
win32	2/25/2015 4:54 PM	File folder	
win64	2/25/2015 4:54 PM	File folder	
activemq	6/5/2014 3:35 PM	File	22 KB
activemq.bat	6/5/2014 3:35 PM	Windows Batch File	5 KB
activemq.jar	6/5/2014 3:17 PM	Executable Jar File	16 KB
activemq-admin	6/5/2014 3:35 PM	File	6 KB
activemq-admin.bat	6/5/2014 3:35 PM	Windows Batch File	5 KB
wrapper.jar	6/5/2014 2:48 PM	Executable Jar File	82 KB

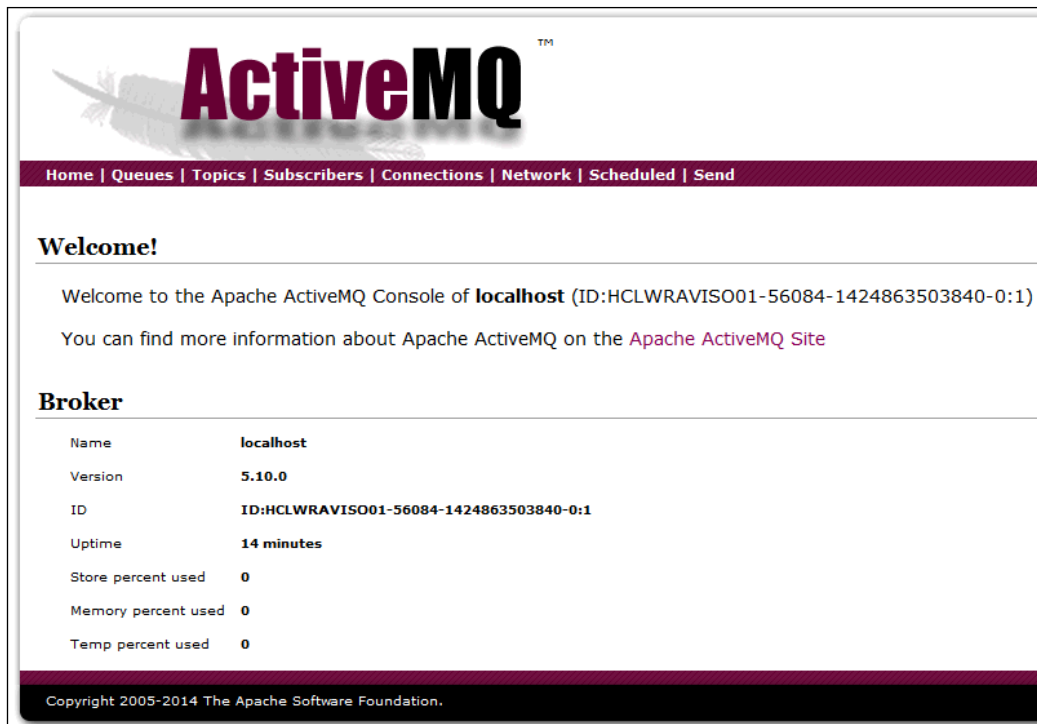
4. Navigate to the `win32` or `win64` folder based on your machine, and open Command Prompt at this location and then run `activemq`, as shown here:

```

rsoni\Desktop\ravi-book\apache-activemq-5.10.0\bin\win64\...\data\kahadb]
jvm 1 | INFO | Apache ActiveMQ 5.10.0 (localhost, ID:HCLWRAVISO01-56084-1424
863503840-0:1) is starting
jvm 1 | INFO | Listening for connections at: tcp://HCLWRAVISO01:61616?maximu
mConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector openwire started
jvm 1 | INFO | Listening for connections at: amqp://HCLWRAVISO01:5672?maximu
mConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector amqp started
jvm 1 | INFO | Listening for connections at: stomp://HCLWRAVISO01:61613?maxi
mumConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector stomp started
jvm 1 | INFO | Listening for connections at: mqtt://HCLWRAVISO01:1883?maximu
mConnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector mqtt started
jvm 1 | INFO | Listening for connections at ws://HCLWRAVISO01:61614?maximumC
onnections=1000&wireFormat.maxFrameSize=104857600
jvm 1 | INFO | Connector ws started
jvm 1 | INFO | Apache ActiveMQ 5.10.0 (localhost, ID:HCLWRAVISO01-56084-1424
863503840-0:1) started
jvm 1 | INFO | For help or more information please see: http://activemq.apac
he.org
jvm 1 | INFO | ActiveMQ WebConsole available at http://0.0.0.0:8161/
jvm 1 | INFO | Initializing Spring FrameworkServlet 'dispatcher'
jvm 1 | INFO | jolokia-agent: No access restrictor found at classpath:/jolok
ia-access.xml, access to all MBeans is allowed

```

We can see in the preceding screenshot, that activemq is run and has provided some information on the console. This MQ can be listed at `tcp://localhost:61616` URL. The admin page URL `http://localhost:8161/admin` provides access to the admin page (username: admin, password: admin):



The Spring bean configuration (Spring.xml)

Create the configuration file `Spring.xml` and define the respective bean definition such as ActiveMQ ConnectionFactory, ActiveMQ queue destination, and JMS template as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jms="http://www.springframework.org/schema/jms"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/
beans/spring-beans.xsd
```

```

context
    http://www.springframework.org/schema/
context/spring-context.xsd
    http://www.springframework.org/schema/jms
spring-jms.xsd
    http://www.springframework.org/schema/jms/
    http://activemq.apache.org/schema/core
activemq-core.xsd">
    http://activemq.apache.org/schema/core/

    <context:component-scan base-package="org.packt.Spring.chapter10.
JMS" />

    <bean id="jmsTemplate"
class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory"
ref="connectionFactory" />
        <property name="defaultDestination" ref="destination" />
    </bean>

    <bean id="connectionFactory" class="org.apache.activemq.
ActiveMQConnectionFactory">
        <property name="brokerURL">
            <value>tcp://localhost:61616</value>
        </property>
    </bean>

    <bean id="destination"
class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg value="myMessageQueue" />
    </bean>

</beans>

```

The Spring Framework supports JMS with the help of the following classes:

- **ActiveMQConnectionFactory:** This will create a JMS ConnectionFactory for ActiveMQ that connects to a remote broker on a specific host name and port
- **ActiveMQQueue:** This will configure the ActiveMQ queue name as in our case myMessageQueue
- **JmsTemplate:** This is a handy abstraction supported by Spring, and it allows us to hide some of the lower-level JMS details while sending a message

MessageSender.java – Spring JMS Template

The MessageSender class is responsible for sending a message to the JMS queue:

```
package org.packt.Spring.chapter10.JMS.Message;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MessageSender {

    @Autowired
    private JmsTemplate jmsTemplate;

    public void send(final Object Object) {
        jmsTemplate.convertAndSend(Object);
    }
}
```

App.java

The App class contains the main method, which calls the send() method to send a message, as shown in the following code snippet:

```
package org.packt.Spring.chapter10.JMS.Main;

import java.util.HashMap;
import java.util.Map;

import org.packt.Spring.chapter10.JMS.Message.MessageSender;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext
;

public class App {
```

```
public static void main(String[] args) {

    ApplicationContext context = new
    ClassPathXmlApplicationContext(
        "Spring.xml");

    MessageSender messageSender = (MessageSender) context
        .getBean("messageSender");

    Map<String, String> message = new HashMap<String,
String>();
    message.put("Hello", "World");
    message.put("city", "Sasaram");
    message.put("state", "Bihar");
    message.put("country", "India");

    messageSender.send(message);

    System.out.println("Message Sent to JMS Queue: " +
message);
}
}
```

Start ActiveMQ

Before you run `App.java`, you need to start ActiveMQ, which allows us to run a broker; it will run ActiveMQ Broker using the out-of-the-box configuration

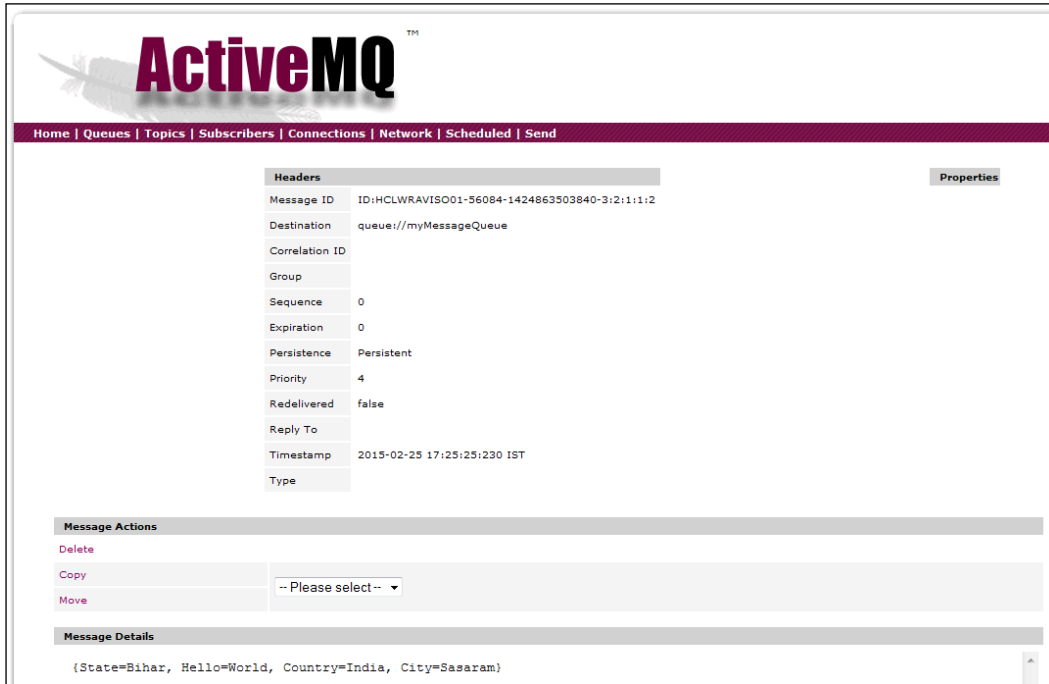
Output

Run `App.java` and get the output on the console as follows:

```
Message Sent to JMS Queue: {state=Bihar, Hello=World,
country=India, city=Sasaram}
```


Monitoring the broker

We can monitor ActiveMQ Broker using the web console by pointing the browser to `http://localhost:8161/admin`. Once `app.java` gets executed, a message will be sent to the JMS queue, as shown in the following screenshot:



The screenshot displays the ActiveMQ web console interface. At the top, the 'ActiveMQ' logo is visible. Below it, a navigation bar contains links: Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send. The main content area is divided into two panels. The left panel, titled 'Headers', lists message properties: Message ID (ID:HCLWRAVISO01-56084-1424863503840-3:2:1:1:2), Destination (queue://myMessageQueue), Correlation ID, Group, Sequence (0), Expiration (0), Persistence (Persistent), Priority (4), Redelivered (false), Reply To, Timestamp (2015-02-25 17:25:25:230 IST), and Type. The right panel, titled 'Properties', is currently empty. Below the headers, the 'Message Actions' section includes links for Delete, Copy, and Move, along with a dropdown menu set to '-- Please select --'. The 'Message Details' section at the bottom shows the message body: `{State=Bihar, Hello=World, Country=India, City=Sasaram}`.

Exception on running App.java

There are chances of getting the error `Could not connect to broker URL exception: tcp://localhost:61616. Reason: java.net.ConnectException: Connection refused: connect`. This exception will come if the message broker service is not up, so make sure that ActiveMQ is running, as shown here:

```
Exception in thread "main"
org.springframework.jms.UncategorizedJmsException: Uncategorized
exception occurred during JMS processing; nested exception is
javax.jms.JMSException: Could not connect to broker URL:
tcp://localhost:61616. Reason: java.net.ConnectException:
Connection refused: connect
    at
org.springframework.jms.support.JmsUtils.convertJmsAccessException
(JmsUtils.java:316)
```

Exercise

Q1. What is a JavaMail API?

Q2. What is message and messaging?

Q3. What is JMS?



The answers to these are provided in *Appendix A, Solution to Exercises*.

Summary

In this chapter, we discussed the e-mail support in Spring and JMS in Spring. We took a look at the JavaMail API and Spring API for JavaMail. Then, we developed a Spring Mail Application. We discussed Spring Java Message Service and understood message, messaging, and JMS components. We took a look at the MOM Service Provider and configured ActiveMQ. We also developed an application to perform messaging using a Spring JMS Template. Then, we discussed the exception on running the application.

In the next chapter, we will go through the solutions of all the exercises given thus far.

Online chapters

Chapter 9, Inversion of Control in Spring – Using Annotation, configures Spring beans and Dependency Injection using annotation. It covers annotation-based Dependency Injection and life cycle annotation. It explains how to reference beans using Spring Expression Language (SpEL), invoke methods using SpEL, and work with operators in SpEL. It also covers the text messages and internationalization provided by Spring, which we will learn to implement in our application. This is an online chapter available at https://www.packtpub.com/sites/default/files/downloads/73680S_Chapter9.pdf.

Chapter 10, Aspect-oriented Programming with Spring, introduces you to aspect-oriented programming. It shows you how and where to apply your aspects in your application using Spring's powerful pointcut mechanism and discusses proxies in the Spring AOP. This is an online chapter available at https://www.packtpub.com/sites/default/files/downloads/73680S_Chapter10.pdf.

Appendix C, Spring Form Tag Library, shows the Spring form tag library provided by the Spring Web MVC framework. The Spring form tag library is a set of tags in the form of a tag library, which is used to construct views (web pages). This is an online appendix available at https://www.packtpub.com/sites/default/files/downloads/73680S_AppendixC.pdf.



Solutions to Exercises

Chapter 1, Introducing the Spring Framework

Q1. What is Spring?

Spring is an open-source framework created by Rod Johnson. He addressed the complexity of enterprise application development, and described a simpler, alternative approach in his book *Expert One-on-One J2EE Design and Development, Wrox*. Spring is a lightweight inversion of control and aspect-oriented container framework. Any Java EE application can benefit from the Spring Framework, in terms of simplicity, loose coupling, and testability.

Spring is modular, allowing you to use only those parts that you need without having to bring in extra complexity. The Spring Framework can be used either for all layer implementations or for the development of particular layer of an application.

Q2. List some of the features of Spring?

The Spring Framework contains following features:

- **Lightweight:** Spring is described as a lightweight framework when it comes to size and transparency.
- **Non-intrusive:** Non-intrusive means that your domain logic code has no dependencies on the framework itself. The Spring Framework is designed to be non-intrusive.
- **Container:** Spring's container is a light-weight container that contains and manages the life cycle and configuration of application objects.

- **Inversion of Control:** IoC is an architectural pattern that describes the Dependency Injection that needs to be done by an external entity, rather than creating the dependencies by the component itself.
- **Aspect-oriented programming:** AOP refers to the programming paradigm that isolates supporting functions from the main program's business logic.
- **JDBC exception handling:** The JDBC abstraction layer of the Spring Framework provides an exception hierarchy.
- **Spring Web MVC framework:** This helps in building robust and maintainable web applications. The Spring Web MVC framework also offers utility classes to handle some of the most common tasks in the web application development.
- **Spring Security:** This provides a declarative security mechanism for Spring-based applications, which is a critical aspect of many applications.

Q3. Explain different modules in the Spring Framework.

The Spring Framework contains following modules:

- Spring Core Module
- Spring AOP Module
- Spring DAO(JDBC) Module
- Spring ORM Module
- Spring Web Module
- Spring Test Module

Chapter 2, Inversion of Control in Spring

Q1. What are Inversion of Control (IoC) and Dependency Injection (DI)?

IoC is a more general concept, and DI is a concrete design pattern. In software engineering, IoC is a programming technique where the assembler object compels object coupling at runtime using static analysis. DI reduces the coupling between objects. DI is a design pattern on which the dependency of object is injected by the framework, rather than created by the object itself.

IoC makes your code more portable, more testable, and more manageable and keeps component configuration, dependencies, and life cycle events outside of the components.

Q2. What are the different types of Dependency Injection in Spring?

In the Spring Framework, DI is used to satisfy the dependencies between objects. It exists in two major types:

- **Constructor Injection:** Constructor-based DI can be accomplished by invoking parameterized constructor. These constructor arguments will be injected during the instantiation of the instance.
- **Setter Injection:** Setter-based DI is the preferred method of Dependency Injection in Spring that can be accomplished by calling setter methods on your bean after invoking a no-argument static factory method or no-argument constructor to instantiate this bean.

Q3. Explain autowiring in Spring. What are the different modes of autowiring.

A Spring container can use five modes of autowiring as follows:

- **no:** By default, Spring bean autowiring is turned off which means that no autowiring is to be performed, and you should use explicit bean reference `ref` for wiring.
- **byName:** This property name is used for this type of autowiring. If the bean property is same as other bean name, autowire it. The setter method is used for this type of autowiring to inject dependency.
- **byType:** This data type is used for this type of autowiring. If the data type bean property is compatible with data type of other bean, autowire it. For this type, only one bean should be configured in configuration file else a fatal exception will be thrown.
- **constructor:** This is similar to autowire `byType`, but here constructor is used for injecting dependency.
- **autodetect:** Autowiring by `autodetect` in Spring is deprecated, and it first tries to autowire by constructor, and if it does not work, then autowire by type.

Q4. Explain different Spring bean scope.

The following list gives the Spring bean scope:

- **Singleton:** Singleton in Spring represents in a particular Spring Container and there is only one instance of bean created in that container that is used across different references.
- **Prototype:** This is a new bean created with every request or reference. For every `getBean()` call, Spring has to do initialization so instead of doing default initialization while a context is being created, it waits for `getBean()` call.

- **Request:** A new bean is created per Servlet request. Spring will be aware of when a new request is happening because it ties well with Servlet APIs and depending on request, Spring creates a new bean.
- **Session:** A new bean is created per session. As long as there is one user accessing in a single session, on each call to `getBean()` will return same instance of bean.
- **Global-session:** This is applicable in portlet context. There will be a global session in an individual portlet session, and a bean can be tied with global session. Here, a new bean is created per global HTTP session.

Chapter 3, DAO and JDBC in Spring

Q1. Explain Spring JDBC packages.

To handle different aspects of JDBC, Spring JDBC is divided into packages, as shown in following table:

Spring JDBC packages	Description
<code>org.springframework.jdbc.core</code>	In the Spring Framework, this package contains the foundations of JDBC classes, which includes Core JDBC Class and <code>JdbcTemplate</code> . It simplifies the database operation using JDBC.
<code>org.springframework.jdbc.datasource</code>	This package contains <code>DataSource</code> implementations and helper classes, which can be used to run the JDBC code outside JEE container.
<code>org.springframework.jdbc.object</code>	In the Spring Framework, this package contains classes that helps in converting the data returned from the database into plain java objects.
<code>org.springframework.jdbc.support</code>	<code>SQLExceptionTranslator</code> is the most important class in this package of the Spring Framework. Spring recognizes the error code used by database using this class, and map error code to higher-level exception.
<code>org.springframework.jdbc.config</code>	This package contains classes that supports JDBC configuration within <code>ApplicationContext</code> of the Spring Framework.

Q2. What is JdbcTemplate?

The JdbcTemplate class instances are thread-safe once configured. A single JdbcTemplate can be configured and injected in multiple DAOs. We can use JdbcTemplate to execute different types of SQL statements. **Data Manipulation Language (DML)** is used to insert, retrieve, update, and delete data in database. The SELECT, INSERT, or UPDATE statements are examples of DML. **Data Definition Language (DDL)** is used to either create or modify the structure of database objects in database. The CREATE, ALTER, and DROP statements are examples of DDL.

Q3. Explain the JDBC operation in Spring.

The single executable unit for performing multiple operations is known as a batch. The batch update operation allows submitting multiple of SQL queries DataSource for processing at once. Submitting multiple SQL queries together, instead of individually improves the performance. The JdbcTemplate includes a support for executing the batch of statements through a JDBC Statement and PreparedStatement. The JdbcTemplate includes two overloaded batchUpdate() methods in support of this feature:

- One for executing a batch of SQL statements using JDBC Statement like:

```
public int[] batchUpdate(String[] sql) throws
    DataAccessException
```
- The other for executing the SQL Statement for multiple times with different parameters using PreparedStatement such as:

```
public int[] batchUpdate(String sql,
    BatchPreparedStatementSetter bPSS) throws
    DataAccessException
```

Chapter 4, Hibernate with Spring

Q1. What is ORM?

ORM is the process of persisting objects in a relational database such as RDBMS. ORM bridges the gap between object and relational schemas, allowing object-oriented application to persist objects directly without having the need for converting object to and from a relational format.

ORM is about mapping object representations to JDBC Statement parameters, and in turn mapping JDBC query results back to object representations. The database columns are mapped to instance fields of domain objects or JavaBeans' properties

Q2. Explain the basics elements of Hibernate architecture.

The basics elements of Hibernate architecture are described in the following sections:

- **Configuration:** The `org.hibernate.cfg.Configuration` class is the basic element of the Hibernate API that allows us to build `SessionFactory`. Configuration can be referred as factory class that can produce `SessionFactory`.
- **SessionFactory:** The `SessionFactory` is created during the startup of the application, and is kept for later use in the application. The `org.hibernate.SessionFactory` interface serves as factory, provides an abstraction to obtain the Hibernate session object. The `SessionFactory` initialization process includes various operations that consume huge resource and extra time, so it is recommended to use single `SessionFactory` per JVM instance.
- **Session:** The `org.hibernate.Session` is an interface between Hibernate system and the application. It is used to get the connection with a database. It is light weight, and is initiated each time an interaction is needed with the database. After we complete the use of `Session`, it has to be closed to release all the resources, such as cached entity objects and JDBC connection.
- **Transaction:** Transactional interface is an optional interface that represents a unit of work with the database, and supported by most of RDBMS. In Hibernate, `Transaction` is handled by the underlying transaction manager.
- **Query:** The `org.hibernate.Query` interface provides an abstraction to execute the Hibernate query and to retrieve the results. The `Query` object represents Hibernate query built using Hibernate Query Language.
- **Criteria:** The `org.hibernate.Criteria` is an interface for using `Criterion` API and is used to create and execute object oriented criteria queries, alternative to HQL or SQL.
- **Persistent:** These classes are the entity classes in an application. Persistent objects are objects that are managed to be in persistent state. Persistent objects are associated with exactly one `org.hibernate.Session`. Once the `org.hibernate.Session` is closed, these objects will be detached and will be free to use in any layer of application.

Q3. What is HQL?

Hibernate Query Language (HQL) is an object-oriented query language that works on the `Persistence` object and their properties, instead of operating on tables and columns. Hibernate will translate the HQL queries into conventional SQL queries during the interaction of database. In HQL, the keywords such as `SELECT`, `FROM`, `WHERE`, and `GROUP BY`, and so on is not case sensitive.

Chapter 5, Spring Security

Q1. What is Spring Security?

The Spring Security framework is the de-facto standards for securing Spring-based applications. Spring Security framework provides security services for enterprise Java software application by handling authentication and authorization. Spring Security handles authentication and authorization at both; the web request level and at method invocation level. Spring Security is a highly customizable and powerful authentication and can access control framework.

Q2. What is authentication and authorization?

Authentication is the process of assuring that a user is the one what user claim to be. Authentication is a combination of identification and verification. The identification can be performed in a number of different ways; for example, as username and password, which can be stored in a database, or LDAP, or CAS (single sign-on protocol) and so on.

Authorization provides access control to the authenticated user. Authorization is the process of ensuring that the authenticated user is allowed to access only those resources which he/she is authorized to use.

Q3. What are the different ways supported by Spring Security for users to log into a web application?

There are multiple ways to be supported by Spring Security for users to log into a web application as follows:

- **HTTP basic authentication:** HTTP basic authentication is supported by Spring Security by processing the basic credentials presented in the header of HTTP request. HTTP basic authentication is generally used with stateless clients who on each request pass their credential.
- **Form-based login Service:** Spring Security supports form-based login service, by providing default login form page for users, to log into the web application.
- **Anonymous login:** An anonymous login service is provided by Spring Security that grants authorities to an anonymous user like the normal user.
- **Remember Me support:** Remember Me login is also supported by Spring Security by remembering the user's identity across multiple browser sessions.

Chapter 6, Spring Testing

Q1. What is the difference between JUnit4 and TestNG?

JUnit and TestNG, both are unit testing frameworks, which look very similar in functionality. Both provide functionalities such as annotation supports, exception test, timeout test, ignore test, and suite test. Whereas, a group test and dependency test is only supported by TestNG. TestNG has the ability to dynamically generate the test data for parameterized test, whereas JUnit cannot. The following is a list of few annotations supported by TestNG and JUnit4:

Feature	TestNG	JUnit4
Test annotation	@Test	@Test
Before the first test method in the current class	@BeforeClass	@BeforeClass
After all the test methods in the current class	@AfterClass	@AfterClass
Before each test method	@BeforeMethod	@Before
After each test method	@AfterMethod	@After
Before all tests in this suite run	@BeforeSuite	-
After all tests in this suite run	@AfterSuite	-
Run before the test	@BeforeTest	-
Run after the test	@AfterTest	-

Q2. What is the difference between unit testing and integration testing?

Unit testing, as the name suggests, is testing of every individual method of the code. It is the method of testing fundamental pieces of your functionality. It is a piece of code written by a software developer to test a specific functionality in the code. Unit tests are more about improving quality and preventing bugs, less about finding them, and are automated using testing frameworks.

Integration testing is the phase of software testing in which individual software modules are combined and tested as a group to ensure that required units are properly integrated and interacted with each other correctly. The purpose of integration testing is to verify functional, performance, and reliability of the code. The integration testing is used to test several units altogether.

Q3. Explain the Spring MVC test Framework.

Spring MVC test framework makes unit testing and integration testing of Spring MVC controller more meaningful by offering first class JUnit support. It helps in testing all aspects of controller method, which has not tested before. It allows us to perform testing in depth without starting a web container.

In order to perform a test of Spring MVC, the Spring `TestContext` framework, along with JUnit or TestNG, make it simple by providing annotation driven unit and integration testing support. The Spring `TestContext` framework can be used using `@RunWith`, `@WebAppConfiguration`, and `@ContextConfiguration` annotation to load Spring configuration, and inject the `WebApplicationContext` to the `MockMvc` for unit and integration test.

Q4. Explain `@ContextConfiguration` and `@WebAppConfiguration`.

The `@ContextConfiguration` annotation is used to set the `ApplicationContext` for test classes, by taking the actual configuration file with the file path. In the following code, we have given the file, so it will take relative path as the root package. We can also give the exact path by specifying the file: prefix. The `@ContextConfiguration` caches the `ApplicationContext` for us, and puts it in a static memory for the entire duration of the test or the test suite. The entire tests executes in the same JVM because of `ApplicationContext` stored in the static memory. If the second JVM is there, it will not have access to the static context, and will result in second `ApplicationContext` to be created.

The `@WebAppConfiguration` annotation is a class-level annotation used to create a web version of the application context in Spring. It is used to denote that the `ApplicationContext`, which is loaded for an integration test and used by that class, is an instance a `WebApplicationContext`. It is important to note that the `@WebAppConfiguration` annotation must be used together with `@ContextConfiguration`.

Q5. Explain `MockMvc` and `@RunWith(SpringJUnit4ClassRunner.class)`.

The `MockMvc` is a key part of Spring MVC Test framework, which can be used to write tests for applications developed using Spring MVC. It is the entry point for Spring MVC Testing. The `MockMvc` mock the entire Spring MVC infrastructure and is created by using the implementations of the `MockMvcBuilder` interface. In order to use Spring MVC testing, the first step is to create an instance of `MockMvc`

The `@RunWith` annotation is a JUnit annotation. It executes the tests in a class annotated with the `@RunWith` annotation, or extends a class annotated with the `@RunWith` annotation by invoking the class passed as the parameter, which means that the tests in annotated class are not executed by the in-built API in the JUnit framework, the runner class used to execute the test case. In order to use Spring's JUnit class runner for running test cases within Spring's `ApplicationContext` environment passed Spring's `SpringJUnit4ClassRunner` class as a parameter.

Chapter 7, Integrating JavaMail and JMS with Spring

Q1. What is a JavaMail API?

A JavaMail API provides a protocol and platform independent framework to provide e-mail support for a Java application. The JavaMail API is a collection of classes and interfaces that comprise an e-mail system. These steps are involved in sending a simple email, using the JavaMail API. They are as follows:

1. Connect to a e-mail server by specifying the username and password, let's say an example; if you want to send an email from `abc@xyz.com`, then you need to connect to the e-mail server of `xyz.com`.
2. Create a message by specifying the recipient's addresses that can include Cc and Bcc addresses as well.
3. Add attachments to the message if any.
4. Transport the message to the e-mail server.

Q2. What is message and messaging?

Message is nothing but bytes of data or information, which are being exchanged between two parties. By taking different specifications; a message can be described in various ways. However, it is nothing but an entity of communication. A message can be used to transfer a piece of information from one application to another application, which may or may not run in the same platform.

Messaging is communication between different applications (in a distributed environment), or system components which are loosely coupled unlike its peers, like TCP sockets, **Remote Method Invocation (RMI)** or CORBA, which is tightly coupled. The advantage of Java messaging includes the ability to integrate different platforms, increase the scalability and reliability of message delivery and reduces the system bottlenecks. Using messaging, we can increase the systems and clients who are consuming and producing the message as much as we want.

Q3. What is JMS?

The JMS, that is, Java Message Service is a Java **Message Oriented Middleware (MOM)** API for sending messages between two or more clients. JMS is a part of Java Enterprise edition. JMS is a broker like a postman, who acts like a mediator between the message sender and receiver.

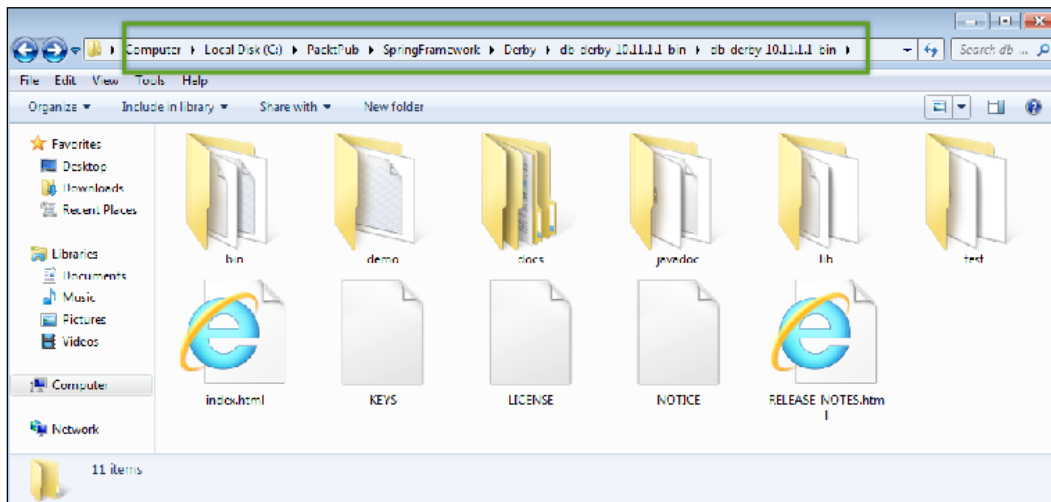
JMS is a specification that describes a common way for Java programs to create, send, and read distributed enterprise messages. It advocates the loosely coupled communication without caring about sender and receiver. It provides asynchronous messaging, that means it doesn't matter whether the sender and the receiver are present at the same time or not. The two systems that are sending or receiving messages need not be up at same time.

B

Setting up the Application Database – Apache Derby

To set up some kind of database running on your development environment, we use Apache Derby database. Apache Derby is a light weight in memory database, which is easy to setup and takes less resources, and is also perfect for testing out new concepts and trying out things that we are doing right now.

To download Apache Derby, hit over the Apache Derby website at http://db.apache.org/derby/derby_downloads.html, and download the latest release. Once the downloaded ZIP file is extracted, we will have some important folder named `bin` and `lib` folder as shown:



The `lib` folder contains the jar that needs to be included in our program when we connect to the Derby database. The `bin` contains programs like `startNetworkServer.bat` and `stopNetworkServer.bat` for database as follows:

```
C:\PacktPub\SpringFramework\Chapter-5\Derby\db-derby-10.11.1.1-bin\db-derby-10.11.1.1-bin\bin>dir
Volume in drive C is Local Disk
Volume Serial Number is 9265-383D

Directory of C:\PacktPub\SpringFramework\Chapter-5\Derby\db-derby-10.11.1.1-bin\
db-derby-10.11.1.1-bin\bin

16/09/2014  07:37 PM    <DIR>          .
16/09/2014  07:37 PM    <DIR>          ..
16/09/2014  07:37 PM             5,740 dblook
16/09/2014  07:37 PM             1,387 dblook.bat
16/09/2014  07:37 PM             2,426 derby_common.bat
16/09/2014  07:37 PM             5,876 ij
16/09/2014  07:37 PM             1,379 ij.bat
16/09/2014  07:37 PM             5,801 NetworkServerControl
16/09/2014  07:37 PM             1,413 NetworkServerControl.bat
16/09/2014  07:37 PM             1,073 setEmbeddedCP
16/09/2014  07:37 PM             1,278 setEmbeddedCP.bat
16/09/2014  07:37 PM             1,079 setNetworkClientCP
16/09/2014  07:37 PM             1,284 setNetworkClientCP.bat
16/09/2014  07:37 PM             1,075 setNetworkServerCP
16/09/2014  07:37 PM             1,273 setNetworkServerCP.bat
16/09/2014  07:37 PM             5,807 startNetworkServer
16/09/2014  07:37 PM             1,397 startNetworkServer.bat
16/09/2014  07:37 PM             5,810 stopNetworkServer
16/09/2014  07:37 PM             1,403 stopNetworkServer.bat
16/09/2014  07:37 PM             5,789 sysinfo
16/09/2014  07:37 PM             1,389 sysinfo.bat
                19 File(s)          52,679 bytes
```

After downloading and extracting the JAR file, the next step is to set environment variable. Derby recommends a couple of environment variables that need to be set.

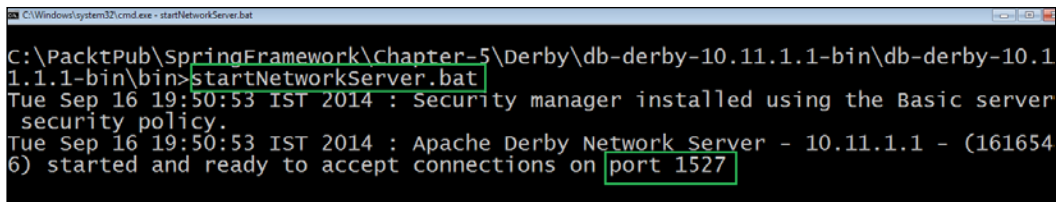
- **DERBY_HOME:** The `DERBY_HOME` environment variable needs to be set to the location where we have extracted our distribution containing the `bin` and `lib` folder.
- **Path:** The second variable that we need to set is path environment variable. We need to set `DERBY_HOME/bin` to path environment variable:

Apache Derby operates in two modes:

- The Network-Server mode
- The Embedded mode

First, we need to start derby as Network Server mode which is similar to all the databases on one machine, and the other machine on the network that can connect to it. Embedded mode is something specific to derby

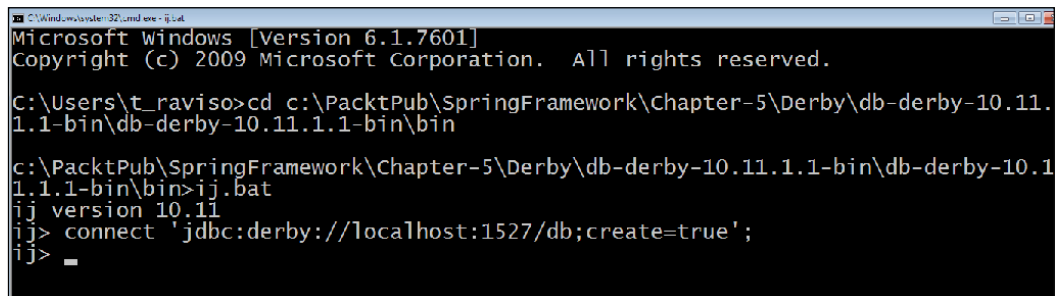
To start derby Network Server mode, we need to run `startNetworkServer.bat` as follows:



```
C:\Windows\system32\cmd.exe - startNetworkServer.bat
C:\PacktPub\SpringFramework\Chapter-5\Derby\db-derby-10.11.1.1-bin\db-derby-10.11.1.1-bin\bin>startNetworkServer.bat
Tue Sep 16 19:50:53 IST 2014 : Security manager installed using the Basic server security policy.
Tue Sep 16 19:50:53 IST 2014 : Apache Derby Network Server - 10.11.1.1 - (1616546) started and ready to accept connections on port 1527
```

In the preceding screenshot, we can see that Apache Derby Network-Server started and ready to accept connection on port 1527, which we can test by using a client to connect to the server. Derby actually comes with a client called `ij.bat`, which can be used to connect to the server to execute queries.

So, we need to have a second command line prompt, which hits same bin directory, and run `ij.bat` to client, and executes the query to connect to the server `connect jdbc:derby://localhost:1527/db;create=true'`; as follows:



```
C:\Windows\system32\cmd.exe - ij.bat
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\t_raviso>cd c:\PacktPub\SpringFramework\Chapter-5\Derby\db-derby-10.11.1.1-bin\db-derby-10.11.1.1-bin\bin
c:\PacktPub\SpringFramework\Chapter-5\Derby\db-derby-10.11.1.1-bin\db-derby-10.11.1.1-bin\bin>ij.bat
ij version 10.11
ij> connect 'jdbc:derby://localhost:1527/db;create=true';
ij> _
```

We can create a table using create query for table employee (as shown in the following screenshot with two column as ID and NAME), and insert values using insert query, and also print data to console using select query, as follows:

```
ij> create table employee (id integer, name char(30));
0 rows inserted/updated/deleted
ij> insert into employee values (03, 'Ravi');
1 row inserted/updated/deleted
ij> select * from employee
> ;
ID          |NAME
-----
3           |Ravi
1 row selected
ij> _
```


Module 2: Spring MVC Beginner's Guide, Second Edition

Chapter 1: Configuring a Spring Development Environment	9
Setting up Java	9
Time for action – installing JDK	10
Time for action – setting up environment variables	11
Configuring a build tool	12
Time for action – installing the Maven build tool	13
Installing a web server	15
Time for action – installing the Tomcat web server	15
Configuring a development environment	16
Time for action – installing Spring Tool Suite	16
Time for action – configuring Maven on STS	17
Time for action – configuring Tomcat on STS	18
Creating our first Spring MVC project	20
Time for action – creating a Spring MVC project in STS	21
Time for action – adding Java version properties in pom.xml	22
What just happened?	24
Spring MVC dependencies	25
Time for action – adding Spring jars to the project	25
What just happened?	27
A jump-start to MVC	29
Time for action – adding a welcome page	29
What just happened?	30
The Dispatcher servlet	31
Time for action – configuring the Dispatcher servlet	32
What just happened?	34
Deploying our project	35
Time for action – running the project	35
Summary	37
Chapter 2: Spring MVC Architecture – Architecting Your Web Store	38
Dispatcher servlet	38
Time for action – examining request mapping	39
What just happened?	40
Pop quiz – request mapping	41
Understanding the Dispatcher servlet configuration	41

Time for action – examining the servlet mapping	42
What just happened?	43
Servlet mapping versus request mapping	43
Pop quiz – servlet mapping	44
Web application context	44
View resolvers	45
Time for action – understanding web application context	46
What just happened?	47
Understanding the web application context configuration	49
Pop quiz – web application context configuration	50
Model View Controller	51
Overview of the Spring MVC request flow	52
The web application architecture	53
The Domain layer	53
Time for action – creating a domain object	54
What just happened?	58
The Persistence layer	59
Time for action – creating a repository object	60
What just happened?	65
The Service layer	70
Time for action – creating a service object	71
What just happened?	73
Have a go hero – accessing the product domain object via a service	75
An overview of the web application architecture	76
Have a go hero – listing all our customers	76
Summary	78
Chapter 3: Control Your Store with Controllers	79
The role of a Controller in Spring MVC	79
Defining a Controller	80
Time for action – adding class-level request mapping	81
What just happened?	82
Default request mapping method	83
Pop quiz – class level request mapping	84
Handler mapping	85
Using URI template patterns	86
Time for action – showing products based on category	86
What just happened?	88
Pop quiz – request path variable	90
Using matrix variables	91
Time for action – showing products based on filters	92
What just happened?	94

Understanding request parameters	96
Time for action – adding a product detail page	97
What just happened?	99
Pop quiz – the request parameter	101
Time for action – implementing a master detail View	101
What just happened?	103
Have a go hero – adding multiple filters to list products	104
Summary	105
Chapter 4: Working with Spring Tag Libraries	106
The JavaServer Pages Standard Tag Library	106
Serving and processing forms	107
Time for action – serving and processing forms	108
What just happened?	112
Have a go hero – customer registration form	116
Customizing data binding	116
Time for action – whitelisting form fields for binding	117
What just happened?	119
Pop quiz – data binding	121
Externalizing text messages	121
Time for action – externalizing messages	122
What just happened?	123
Have a go hero – externalizing all the labels from all the pages	124
Summary	124
Chapter 5: Working with View Resolver	125
Resolving Views	125
RedirectView	127
Time for action – examining RedirectView	127
What just happened?	128
Pop quiz – RedirectView	129
Flash attribute	130
Serving static resources	131
Time for action – serving static resources	131
What just happened?	132
Pop quiz – static view	133
Time for action – adding images to the product detail page	134
What just happened?	136
Multipart requests in action	136
Time for action – adding images to a product	137
What just happened?	139
Have a go hero – uploading product user manuals to the server	142
Using ContentNegotiatingViewResolver	142

Time for action – configuring ContentNegotiatingViewResolver	143
What just happened?	145
Working with HandlerExceptionResolver	147
Time for action – adding a ResponseStatus exception	148
What just happened?	149
Time for action – adding an exception handler	150
What just happened?	153
Summary	154
Chapter 6: Internalize Your Store with Interceptor	155
Working with interceptors	156
Time for action – configuring an interceptor	156
What just happened?	158
Pop quiz – interceptors	160
LocaleChangeInterceptor – internationalization	161
Time for action – adding internationalization	162
What just happened?	165
Have a go hero – fully internationalize the product details page	167
Mapped interceptors	167
Time for action – mapped intercepting offer page requests	168
What just happened?	172
Summary	174
Chapter 7: Incorporating Spring Security	175
Using Spring Security	175
Time for action – authenticating users based on roles	176
What just happened?	182
Pop quiz – Spring Security	186
Have a go hero – play with Spring Security	186
Summary	186
Chapter 8: Validate Your Products with a Validator	187
Bean Validation	187
Time for action – adding Bean Validation support	188
What just happened?	191
Have a go hero – adding more validation in the Add new product page	195
Custom validation with JSR-303/Bean Validation	195
Time for action – adding Bean Validation support	195
What just happened?	198
Have a go hero – adding custom validation to a category	200
Spring validation	200
Time for action – adding Spring validation	201
What just happened?	203

Time for action – combining Spring validation and Bean Validation	204
What just happened?	207
Have a go hero – adding Spring validation to a product image	209
Summary	210
Chapter 9: Give REST to Your Application with Ajax	211
Introduction to REST	211
Time for action – implementing RESTful web services	212
What just happened?	227
Time for action – consuming REST web services	230
What just happened?	235
Handling web services in Ajax	236
Time for action – consuming REST web services via Ajax	236
What just happened?	242
Summary	245
Chapter 10: Float Your Application with Web Flow	246
Working with Spring Web Flow	246
Time for action – implementing the order processing service	247
What just happened?	260
Time for action – implementing the checkout flow	261
What just happened?	264
Understanding flow definitions	264
Understanding checkout flow	265
Pop quiz – web flow	270
Time for action – creating Views for every view state	271
What just happened?	284
Have a go hero – adding a decision state	286
Summary	287
Chapter 11: Template with Tiles	288
Enhancing reusability through Apache Tiles	288
Time for action – creating Views for every View state	289
What just happened?	295
Pop quiz – Apache Tiles	298
Summary	298
Chapter 12: Testing Your Application	299
Unit testing	300
Time for action – unit testing domain objects	300
What just happened?	302
Have a go hero – adding tests for Cart	304
Integration testing with the Spring Test context framework	304

Time for action – testing product validator	304
What just happened?	307
Time for action – testing product Controllers	309
What just happened?	311
Time for action – testing REST Controllers	312
What just happened?	314
Have a go hero – adding tests for the remaining REST methods	315
Summary	316
Thank you readers!	316
Chapter 13: Using the Gradle Build Tool	317
Installing Gradle	317
The Gradle build script for your project	318
Understanding the Gradle script	319
Chapter 14: Pop Quiz Answers	321
Chapter 2, Spring MVC Architecture – Architecting Your Web Store	321
Chapter 3, Control Your Store with Controllers	321
Chapter 4, Working with Spring Tag Libraries	323
Chapter 5, Working with View Resolver	323
Chapter 6, Internalize Your Store with Interceptor	323
Chapter 7, Incorporating Spring Security	324
Chapter 10, Float Your Application with Web Flow	325
Chapter 11, Template with Tiles	325
Index	326

Module 2

Spring MVC Beginner's Guide, Second Edition

Unleash the power of the latest Spring MVC 4.x to develop a complete application

1

Configuring a Spring Development Environment

In this chapter, we are going to take a look at how we can create a basic Spring MVC application. In order to develop a Spring MVC application, we need some prerequisite software and tools. First, we are going to see how to install all the prerequisites that are required to set up our development environment so that we can start developing the application.

The setup and installation steps given here are for Windows 10 operating systems, but don't worry, as the steps may change only slightly for other operating systems. You can always refer to the respective tools and software vendor's websites to install them in other operating systems. In this chapter, we will learn to set up Java and configure the Maven build tool, install the Tomcat web server, install and configure the Spring Tool Suite, and create and run our first Spring MVC project.

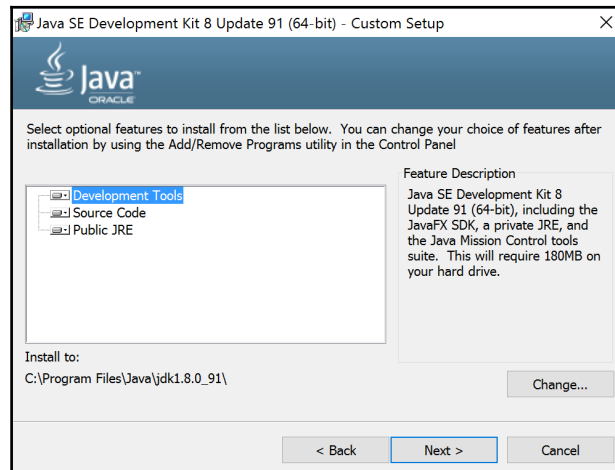
Setting up Java

Obviously, the first thing that we need to do is to install Java. The more technical name for Java is **Java Development Kit (JDK)**. JDK includes a Java compiler (javac), a Java virtual machine, and a variety of other tools to compile and run Java programs.

Time for action – installing JDK

We are going to use Java 8, which is the latest and greatest version of Java, but Java 6 or any higher version is also sufficient to complete this chapter, but I strongly recommend you use Java 8 since in later chapters of this book we may use some of the Java 8 features such as, streams and lambda expressions. Let's take a look at how we can install JDK on a Windows operating system:

1. Go to the Java SE download page on the Oracle website at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
2. Click on the **Java Platform (JDK) 8u91/8u92** download link; this will take you to the license agreement page. Accept the license agreement by selecting the radio button option.
3. Now, click on the listed download link that corresponds to your Windows operating system architecture; for instance, if your operating system is of type 32 bit, click on the download link that corresponds to **Windows x86**. If your operating system is of type 64 bit, click on the download link that corresponds to **Windows x64**.
4. Now it will start downloading the installer. Once the download is finished, go to the downloaded directory and double-click on the installer. This will open up a wizard window. Just click through the next buttons in the wizard, leaving the default options alone, and click on the **Close** button at the end of the wizard:



JDK installation wizard



Additionally, a separate wizard also prompts you to install **Java Runtime Environment (JRE)**. Go through that wizard as well to install JRE in your system.

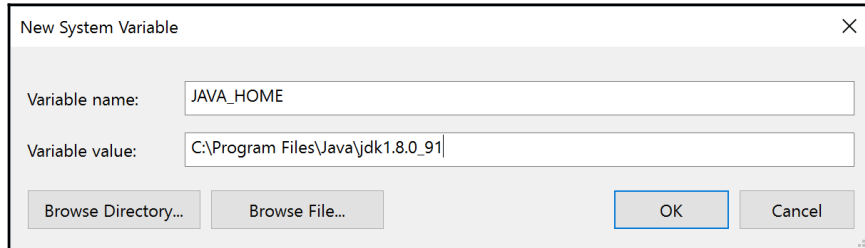
5. Now you can see the installed JDK directory in the default location; in our case, the default location is `C:\Program Files\Java\jdk1.8.0_60`.

Time for action – setting up environment variables

After installing JDK, we still need to perform some more configurations to use Java conveniently from any directory on our computer. By setting up the environment variables for Java in the Windows operating system, we can make the Java compiler and tools accessible from anywhere in the file system:

1. Navigate to **Start Menu | Settings | System | About | System info | Advanced system settings**.
2. A **System Properties** window will appear; in this window, select the **Advanced** tab and click on the **Environment Variables** button to open the **Environment Variables** window.
3. Now, click on the **New** button in the **System variables** panel and enter `JAVA_HOME` as the variable name and enter the installed JDK directory path as the variable value; in our case, this would be `C:\Program Files\Java\jdk1.8.0_91`. If you do not have proper rights for the operating system, you will not be able to edit **System variables**; in that case, you can create the `JAVA_HOME` variable under the **User variables** panel.
4. Now, in the same **System variables** panel, double-click on the path variable entry; an **Edit System Variable** window will appear.
5. Edit **Variable value of Path** by clicking the new button and enter the following text `%JAVA_HOME%\bin` as the value.

If you are using a Windows operating system prior to version 10, edit the path variable carefully; you should only append the text at the end of an existing path value. Don't delete or disturb the existing values; make sure you haven't missed the ; (semi-colon) delimited mark as the first letter in the text that you append:



6. Now click on the **OK** button.

Now we have installed JDK in our computer. To verify whether our installation has been carried out correctly, open a new command window, type `java -version`, and press *Enter*; you will see the installed version of Java compiler on the screen:

```
C:\Users\Amuthan>java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b15, mixed mode)
```

Configuring a build tool

Building a java software project typically includes some activities as follows:

- Compiling all the source code
- Packaging the compiled code into a JAR or WAR archive file
- Deploying the packaged archives files on a server

Manually performing all these tasks is time-consuming and is prone to errors. Therefore, we take the help of a build tool. A build tool is a tool that automates everything related to building a software project, from compiling to deploying.

Time for action – installing the Maven build tool

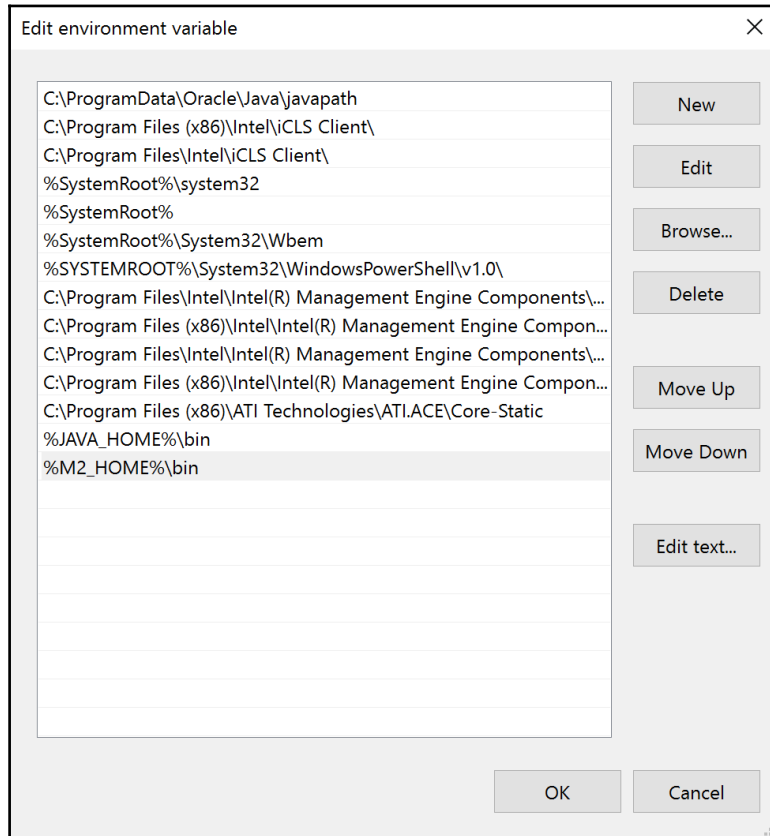
There are other build tools that are available for building Java projects such as Gradle and Ant. We are going to use Maven as our build tool. Let's take a look at how we can install Maven:

1. Go to Maven's download page at <http://maven.apache.org/download.cgi>.
2. Click on the **apache-maven-3.3.9-bin.zip** download link and start the download.



At the time of writing this book, the latest Maven version is 3.3.9; you can literally use any version of Maven after 3.0 to complete this book.

3. Once the download is finished, go to the downloaded directory and extract the .zip file into a convenient directory of your choice.
4. Now we need to create one more environment variable called `M2_HOME` in a way that is similar to the way in which we created `JAVA_HOME`. Enter the extracted Maven zip directory's path as the value for the `M2_HOME` environment variable.
5. Finally, append the `M2_HOME` variable to the `PATH` environment variable as well. Double-click on the path variable and click on the **New** button to enter `%M2_HOME%\bin` as the value.



Setting the M2 environment variable

6. Now we have installed the Maven build tool in our computer. To verify whether our installation has been carried out correctly, we need to follow steps that are similar to the Java installation verification. Open a new command window, type `mvn -version` and press *Enter*; you will see the following details of the Maven version:

```
C:\Users\Amuthan>mvn -version
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5;
2015-11-10T10:41:47-06:00)
Maven home: C:\Program Files\apache-maven-3.3.9
Java version: 1.8.0_91, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_91\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "dos"
```

Installing a web server

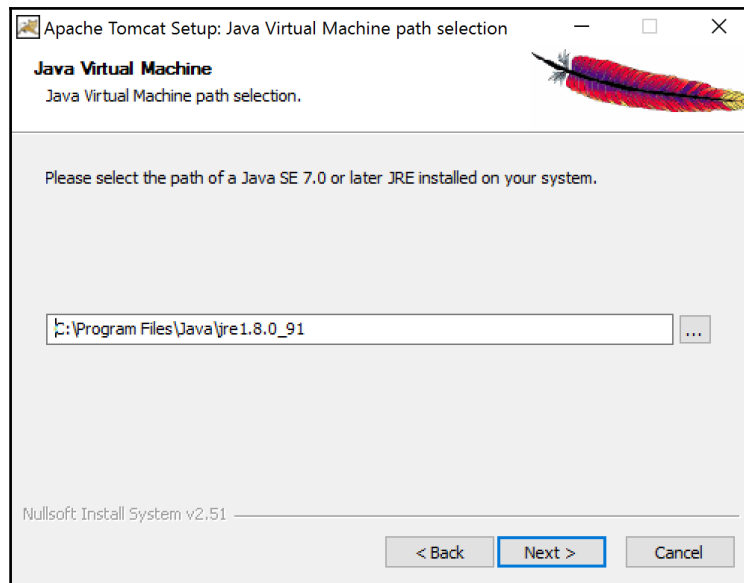
So far, we have seen how to install JDK and Maven. Using these tools, we can compile the Java source code into the `.class` files and package these `.class` files into the `.jar` or `.war` archives. However, how do we run our packaged archives? For this, we take the help of a web server, which will host our packaged archives as a running application.

Time for action – installing the Tomcat web server

Apache Tomcat is a popular Java web server and servlet container. We are going to use Apache Tomcat Version 8.0, which is the latest, but you can even use version 7.0. Let's take a look at how we can install the Tomcat web server:

1. Go to the Apache Tomcat home page at <http://tomcat.apache.org/>.
2. Click on the **Tomcat 8**. download link; it will take you to the download page.
3. Click on the **32-bit/64-bit Windows Service Installer (pgp, md5, sha1)** link; it will start downloading the installer.
4. Once the download is finished, go to the downloaded directory and double-click on the installer; this will open up a wizard window.
5. Just click through the **Next** buttons in the wizard, leaving the default options alone, and click on the **Finish** button at the end of the wizard. Note that before clicking on the **Finish** button, just ensure that you have unchecked **Run Apache Tomcat** checkbox.

Installing Apache Tomcat with the default option works successfully only if you have installed Java in the default location. Otherwise, you have to correctly provide the JRE path according to the location of your Java installation during the installation of Tomcat, as shown in the following screenshot:



The Java runtime selection for the Tomcat installation

Configuring a development environment

We installed Java and Maven to compile and package our Java source code and installed Tomcat to deploy and run our application. So now we have to start write Spring MVC code so that we can compile, package, and run the code. We can use any simple text editor on our computer to write our code, but that won't help us much with features like finding syntax errors as we type, auto-suggesting important key words, syntax highlighting, easy navigation, and so on.

An **integrated development environment (IDE)** can help us with these features to develop the code faster and error free. We are going to use **Spring Tool Suite (STS)** as our IDE.

Time for action – installing Spring Tool Suite

STS is the best Eclipse-powered development environment to build Spring applications. Let's take a look at how we can install STS:

1. Go to the STS download page at <http://spring.io/tools/sts/all>.
2. Click on the STS zip link to download the zip file that corresponds to your Windows operating system architecture type (32 bit or 62 bit); this will start the download of the zip file. The STS stable release version at the time of this writing is the STS 3.7.3.RELEASE based on Eclipse 4.6.
3. Once the download is finished, go to the downloaded directory and extract the .zip file into a convenient directory of your choice.
4. Open the extracted `sts-bundle` directory, you will find a directory called `sts-3.7.3.RELEASE`. Just open that directory and create a desktop shortcut for the `STS.exe`

We have almost installed all the tools and software required to develop a Spring MVC application; so now, we can create our Spring MVC project on STS. However, before jumping into creating a project, we need to perform the following two final configurations on our STS in order to use STS effectively:

1. Configuring Maven on STS
2. Configuring Tomcat on STS

The aforementioned settings are just a one-time configuration that you need to set up on your newly installed STS; you need not perform this configuration every time you open STS

Time for action – configuring Maven on STS

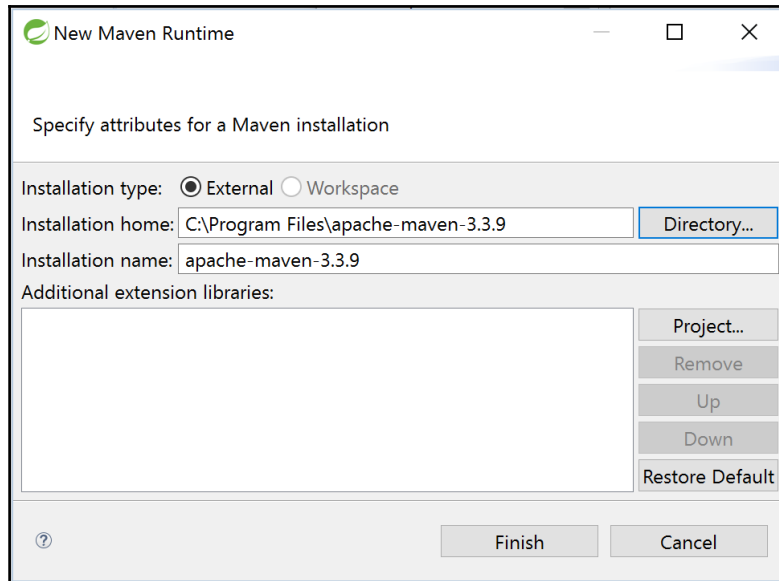
To build our projects, STS uses Maven build tool internally. But we have to tell STS where Maven has been installed so that it can use the Maven installation to build our projects. Let's take a look at how we can configure Maven on STS:

1. Open STS if it is not already open.



When you open STS for the very first time after installing, it will ask you to provide a workspace location. This is because when you create a project on STS, all your project files will be created under this location only. Provide a workspace directory path as you wish and click on the **OK** button.

2. Navigate to **Window | Preferences | Maven | Installations**.
3. On the right-hand side, you can see the **Add** button to locate Maven's installation.
4. Click on the **Add** button and choose our Maven's installed directory, and then click on the **Finish** button, as shown in the following screenshot:



Selecting Maven's location during the Maven configuration on STS

5. Now don't forget to select the newly added Maven installation as your default Maven installation by selecting the checkbox;
6. Click on the **OK** button in the **Preferences** window and close it.

Time for action – configuring Tomcat on STS

As mentioned previously, we can use the Tomcat web server to deploy our application, but we have to inform STS about the location of the Tomcat container so that we can easily deploy our project in to Tomcat. Let's configure Tomcat on STS:

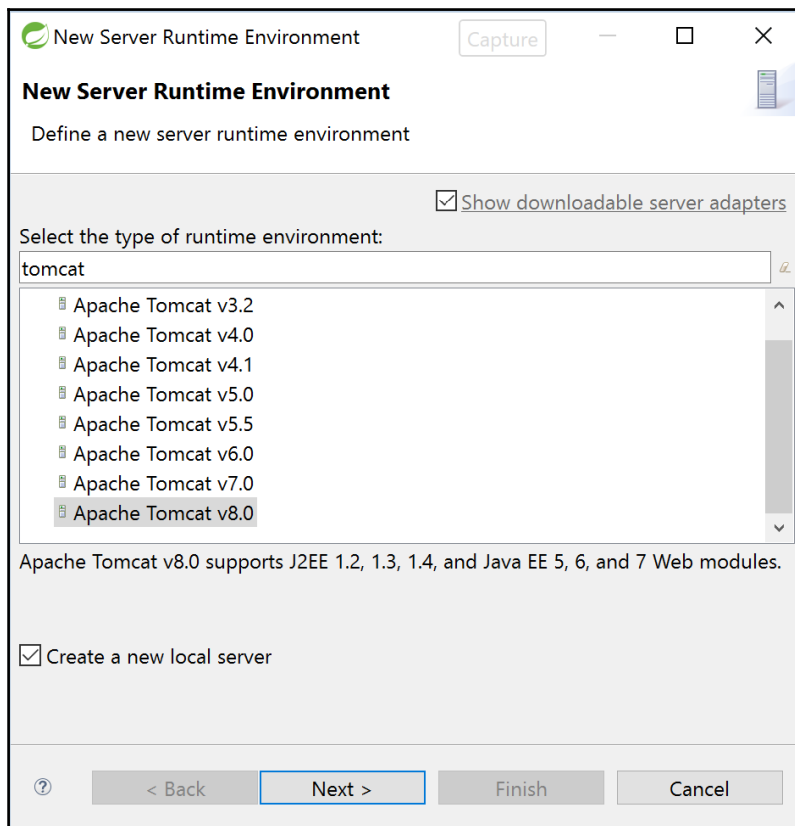
1. Open STS from the desktop icon, if it is not already open.
2. Go to the menu bar and navigate to **Window | Preferences | Server | Runtime Environments**.

3. You can see the available servers listed on the right panel. Now click on the **Add** button to add our Tomcat web server.



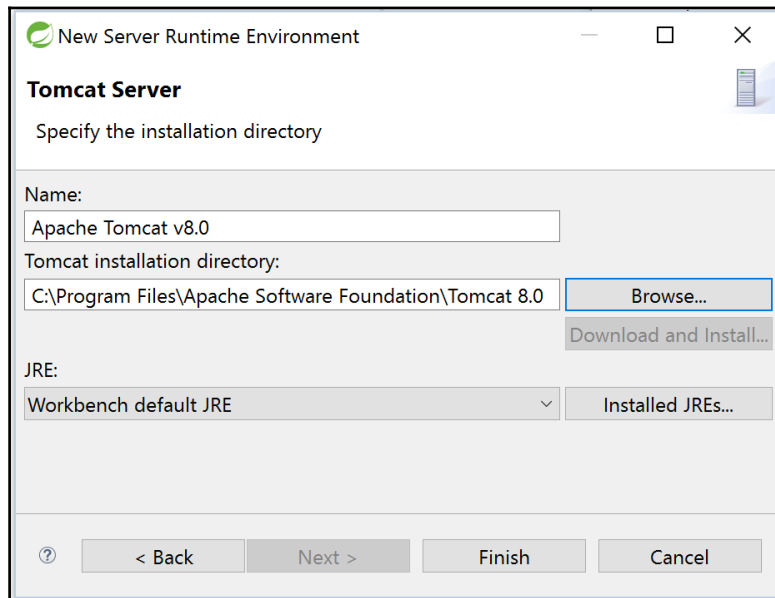
You may also see **Pivotal tc Server Developer Edition (Runtime) v3.1** listed under the available servers, which comes along with the STS installation. Although STS might come with an internal Pivotal tc Server, we chose to use the Tomcat web server as our server runtime environment because of its popularity.

4. A wizard window will appear; type `tomcat` in the **Select the type of runtime environment:** text box, and a list of available Tomcat versions will be shown. Just select **Tomcat v8.0** and select the **Create a new local server** checkbox. Finally, click on the **Next** button, as shown in the following screenshot:



Selecting the server type during the Tomcat configuration on STS

5. In the next window, click on the **Browse** button and locate Tomcat's installed directory, and then click on the **OK** button. You can find Tomcat's installed directory under `C:\Program Files\Apache Software Foundation\Tomcat 8.0` if you have installed Tomcat in the default location. Then, click on the **Finish** button, as shown in the following screenshot:



Selecting the Tomcat location during the Tomcat configuration on STS

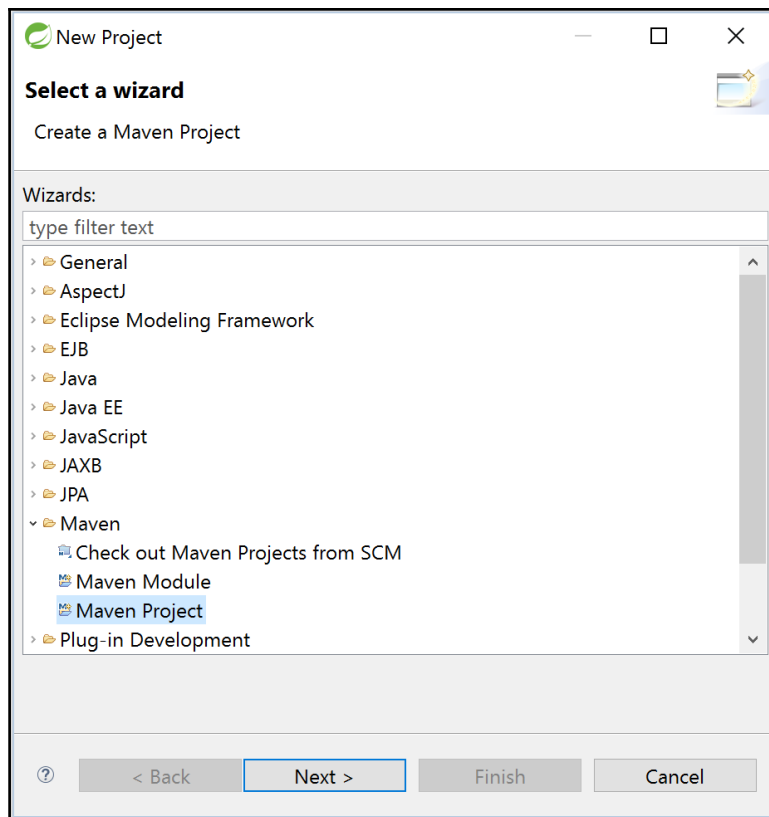
Creating our first Spring MVC project

So far, we have seen how we can install all the prerequisite tools and software. Now we are going to develop our first Spring MVC application using STS. STS provides an easy-to-use project template. Using these templates, we can quickly create our project directory structures without much problem.

Time for action – creating a Spring MVC project in STS

Let's create our first Spring MVC project in STS:

1. In STS, navigate to **File | New | Project**; a **New Project** wizard window will appear.
2. Select **Maven Project** from the list and click on the **Next** button, as shown in the following screenshot:



Maven project's template selection

3. Now, a **New Maven Project** dialog window will appear; just select the checkbox that has the **Create a simple project (skip archetype selection)** caption and click on the **Next** button.

- The wizard will ask you to specify artifact-related information for your project; just enter **Group Id** as `com.packt` and **Artifact Id** as `webstore`. Then, select **Packaging** as `war` and click on the **Finish** button, as shown in the following screenshot:

The screenshot shows the 'New Maven Project' dialog box. The 'Artifact' section is expanded, showing 'Group Id' as 'com.packt', 'Artifact Id' as 'webstore', 'Version' as '0.0.1-SNAPSHOT', and 'Packaging' as 'war'. The 'Parent Project' section is also visible, with 'Group Id' and 'Artifact Id' fields. The 'Finish' button is highlighted in blue.

Specifying artifact-related information during the project creation

Time for action – adding Java version properties in `pom.xml`

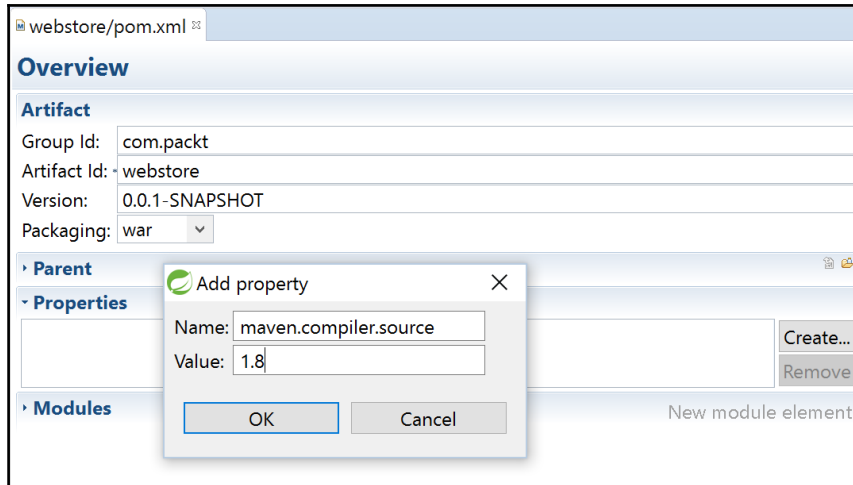
We have successfully created a basic project, but we need to perform one small configuration in our `pom.xml` file, that is, telling Maven to use Java Version 8 while compiling and building our project. How do we tell Maven to do this? Simply add two property entries in `pom.xml`. Let's do the following:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project itself.
2. You will see some tabs at the bottom of the `pom.xml` file. Select the **Overview** tab.



If you do not see these tabs, then right-click on `pom.xml`, select the **Open With...** option from the context menu, and choose **Maven POM editor**.

3. Expand the **Properties** accordingly and click on the **Create** button.
4. Now, an **Add property** window will appear; enter **Name** as `maven.compiler.source` and **Value** as `1.8`, as shown in the following screenshot:



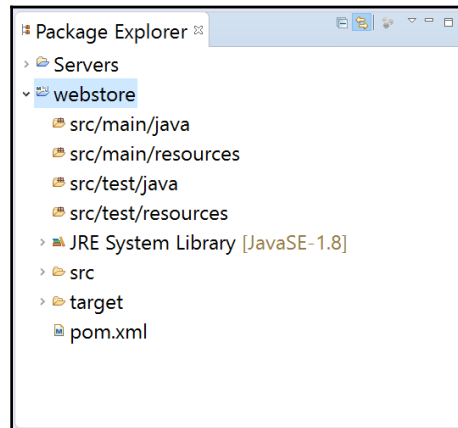
Adding the Java compiler version properties to POM

5. Similarly, create one more property with **Name** as `maven.compiler.target` and **Value** as `1.8`.
6. Finally, save `pom.xml`.

What just happened?

We just created the basic project structure. Any Java project follows a certain directory structure to organize its source code and resources. Instead of manually creating the whole directory hierarchy by ourselves, we just handed over that job to STS. By collecting some basic information about our project, such as **Group Id**, **Artifact Id**, and the **Packaging** style, from us, it is clear that STS is smart enough to create the whole project directory structure with the help of the Maven. Actually, what is happening behind the screen is that STS is internally using Maven to create the project structure.

We want our project to be deployable in any servlet container-based web server, such as Tomcat or Jetty, and that's why we selected the **Packaging** style as `war`. Finally, you will see the project structure in **Package Explorer**, as shown in the following screenshot:



The project structure of the application

If you encounter a maven error on your pom file saying **web.xml is missing and `<failOnMissingWebXml>` is set to true**, then it means it is expecting a `web.xml` file in your Maven project because it is a web application, as we have chosen packaging as `war`. However, nowadays in web applications `web.xml` file is optional. Add the following configuration in your `pom.xml` within `<project>` tag to fix the error:



```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
```



```
<version>2.6</version>
<configuration>
  <failOnMissingWebXml>false</failOnMissingWebXml>
</configuration>
</plugin>
</plugins>
</build>
```

Spring MVC dependencies

As we are going to use Spring MVC APIs heavily in our project, we need to add the Spring jars in our project to make use of it in our development. As mentioned previously, Maven will take care of managing dependency jars and packaging the project.

Time for action – adding Spring jars to the project

Let's take a look at how we can add the Spring-related jars via the Maven configuration:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project itself.
2. You will see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab.



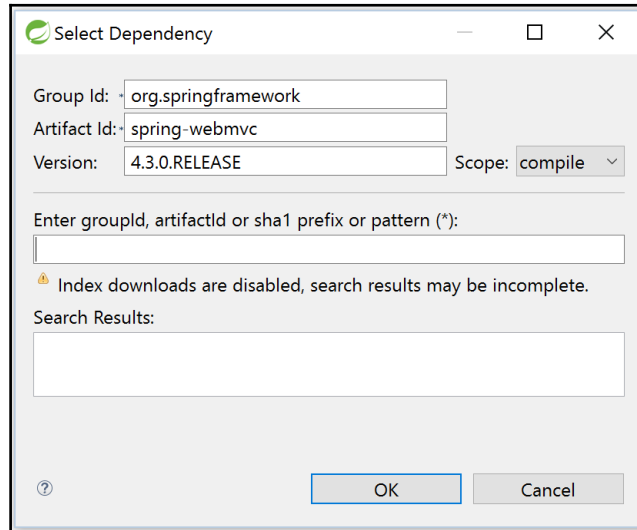
If you do not see these tabs, then right-click on `pom.xml`, select the **Open With...** option from the context menu, and choose **Maven POM editor**.

3. Click on the **Add** button in the **Dependencies** section.



Don't get confused with the **Add** button of the **Dependencies Management** section. You should choose the **Add** button from the left-hand side of the panel.

4. A **Select Dependency** window will appear; enter **Group Id** as `org.springframework`, **Artifact Id** as `spring-webmvc`, and **Version** as `4.3.0.RELEASE`. Select **Scope** as **compile** and then click on the **OK** button, as shown in the following screenshot:



Adding the spring-webmvc dependency

5. Similarly, add the dependency for **JavaServer Pages Standard Tag Library (JSTL)** by clicking on the same **Add** button; this time, enter **Group Id** as `javax.servlet`, **Artifact Id** as `jstl`, **Version** as `1.2`, and select **Scope** as **compile**.
6. Finally, add one more dependency for **servlet-api**; repeat the same step with **Group Id** as `javax.servlet`, **Artifact Id** as `javax.servlet-api`, and **Version** as `3.1.0`, but this time, select **Scope** as **provided** and then click on the **OK** button.
7. As a last step, don't forget to save the `pom.xml` file.

What just happened?

In the Maven world, `pom.xml` (Project Object Model) is the configuration file that defines the required dependencies. While building our project, Maven will read that file and try to download the specified jars from the Maven central binary repository. You need Internet access in order to download jars from Maven's central repository. Maven uses an addressing system to locate a jar in the central repository, which consists of **Group Id**, **Artifact Id**, and **Version**.

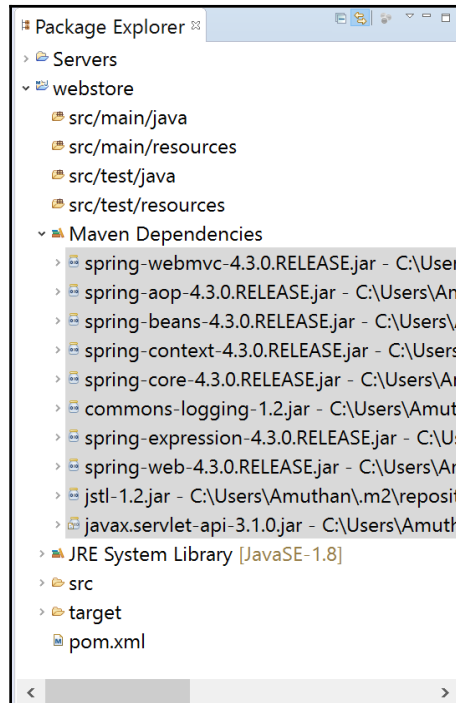
Every time we add a dependency, an entry will be made within the `<dependencies>` `</dependencies>` tags in the `pom.xml` file. For example, if you go to the **pom.xml** tab after finishing step 3, you will see an entry for `spring-webmvc` as follows with in `<dependencies>` `</dependencies>` tag:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.2.2.RELEASE</version>
</dependency>
```

We added the dependency for `spring-mvc` in step 3, and in step 4, we added the dependency for JSTL. JSTL is a collection of useful **JSP (JavaServer Pages)** tags that can be used to write JSP pages easily. Finally, we need a `javax.servlet-api.jar` in order to use servlet-related code; this is what we added in step 5.

However, there is a little difference in the scope of the `javax.servlet-api` dependency compared to the other two dependencies. We only need `javax.servlet-api` while compiling our project. While packaging our project as `war`, we don't want to ship the `javax.servlet-api.jar` as part of our project. This is because the Tomcat web server would provide the `javax.servlet-api.jar` while deploying our project. This is why we selected the **Scope** as **provided** for `javax.servlet-api`.

After finishing step 6, you will see all the dependent jars configured in your project, as shown in the following screenshot, under the **Maven Dependencies** library:



Showing Maven dependencies after configuring POM

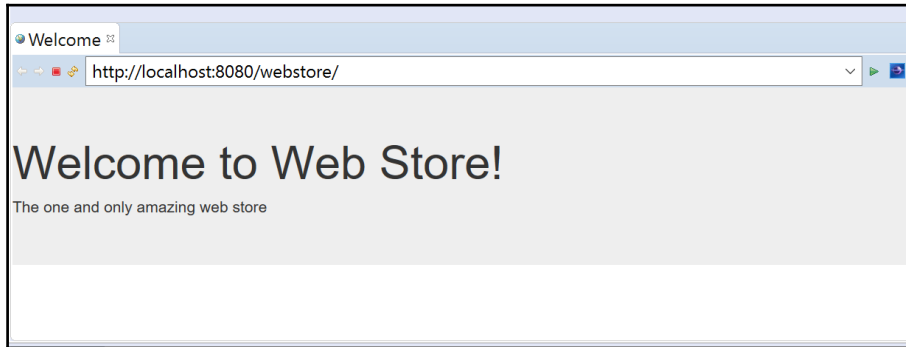
We added only three jars as our dependencies, but if you look in our Maven dependency library list, you will see more than three jar entries. Can you guess why? What if our dependent jars have a dependency on other jars and so on?

For example, our `spring-mvc.jar` is dependent on the `spring-core`, `spring-context`, and `spring-aop` jars, but we have not specified those jars in our `pom.xml` file; this is called **transitive dependencies** in the Maven world. In other words, we can say that our project is transitively dependent on these jars. Maven will automatically download all these transitive dependent jars; this is the beauty of Maven. It will take care of all the dependency management automatically; we need to inform Maven only about the first-level dependencies.

A jump-start to MVC

We have created our project and added all the required jars, so we are ready to code. We are going to incrementally build an online web store throughout this book, chapter by chapter. As a first step, let's create a home page in our project to welcome our customers.

Our aim is simple; when we enter the URL `http://localhost:8080/webstore/` on the browser, we would like to show a welcome page that is similar to the following screenshot:



Showing the welcome page of the web store

We are going to take a deep look at each concept in detail in the upcoming chapters. As of now, our aim is to have quick hands-on experience of developing a simple web page using Spring MVC. So don't worry if you are not able to understand some of the code here.

Time for action – adding a welcome page

To create a welcome page, we need to execute the following steps:

1. Create a `webapp/WEB-INF/jsp/` folder structure under the `src/main/` folder; create a JSP file called `welcome.jsp` under the `src/main/webapp/WEB-INF/jsp/` folder, and add the following code snippets into it and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
```

```
<meta http-equiv="X-UA-Compatible"
      content="IE=edge">
<meta name="viewport" content="width=device-width,
      initial-scale=1">
<title>Welcome</title>
<link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/
      bootstrap/3.3.5/css/bootstrap.min.css">
</head>
<body>
    <div class="jumbotron">
        <h1> ${greeting} </h1>
        <p> ${tagline} </p>
    </div>
</body>
</html>
```

2. Create a class called `HomeController` under the `com.packt.webstore.controller` package in the source directory `src/main/java` and add the following code into it:

```
package com.packt.webstore.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

    @RequestMapping("/")
    public String welcome(Model model) {
        model.addAttribute("greeting", "Welcome to Web Store!");
        model.addAttribute("tagline", "The one and only amazing
        web store");

        return "welcome";
    }
}
```

What just happened?

In step 1, we just created a JSP file; the important thing we need to notice here is the `<h1>` tag and the `<p>` tag. Both the tags have some expression that is surrounded by curly braces and prefixed by the `$` symbol:

```
<h1> ${greeting} </h1>
<p> ${tagline} </p>
```

So, what is the meaning of `${greeting}`? It means that `greeting` is a kind of variable; during rendering of this JSP page, the value stored in the `greeting` variable will be shown in the header 1 style, and similarly, the value stored in the `tagline` variable will be shown as a paragraph.

So now, the next question is where we will assign values to those variables arises. This is where the controller will be of help; within the `welcome` method of the `HomeController` class, take a look at the following lines of code:

```
model.addAttribute("greeting", "Welcome to Web Store!");
model.addAttribute("tagline", "The one and only amazing web store");
```

You can observe that the two variable names, `greeting` and `tagline`, are passed as a first parameter of the `addAttribute` method and the corresponding second parameter is the value for each variable. So what we are doing here is simply putting two strings, "Welcome to Web Store!" and "The one and only amazing web store", into the model with their corresponding keys as `greeting` and `tagline`. As of now, simply consider the fact that `model` is a kind of map data structure.



Folks with knowledge of servlet programming can consider the fact that `model.addAttribute` works exactly like `request.setAttribute`.

So, whatever value we put into the model can be retrieved from the view (JSP) using the corresponding key with the help of the `${}` placeholder expression. In our case, `greeting` and `tagline` are keys.

The Dispatcher servlet

We put values into the model, and we created the view that can read those values from the model. So, the Controller acts as an intermediate between the View and the Model; with this, we have finished all the coding part required to present the welcome page. So will we be able to run our project now? No. At this stage, if we run our project and enter the URL `http://localhost:8080/webstore/` on the browser, we will get an **HTTP Status 404** error. This is because we have not performed any servlet mapping yet.



So what is servlet mapping? Servlet mapping is a configuration of mapping a servlet to a URL or URL pattern. For example, if we map a pattern like `/status/*` to a servlet, all the HTTP request URLs starting with a text `status` such as `http://example.com/status/synopsis` or `http://example.com/status/complete?date=today` will be mapped to that particular servlet. In other words, all the HTTP requests that carry the URL pattern `/status/*` will be handed over to the corresponding mapped servlet class.

In a Spring MVC project, we must configure a servlet mapping to direct all the HTTP requests to a single front servlet. The front servlet mapping is a design pattern where all requests for a particular web application are directed to the same servlet. This pattern is sometimes called as **Front Controller Pattern**. By adapting the Front Controller design, we make front servlet have total control over the incoming HTTP request so that it can dispatch the HTTP request to the desired controller.

One such front servlet given by Spring MVC framework is the Dispatcher servlet (`org.springframework.web.servlet.DispatcherServlet`). We have not configured a Dispatcher servlet for our project yet; this is why we get the **HTTP Status 404** error if we run our project now.

Time for action – configuring the Dispatcher servlet

The Dispatcher servlet is what examines the incoming HTTP request and invokes the right corresponding controller method. In our case, the `welcome` method from the `HomeController` class needs to be invoked if we make an HTTP request by entering the `http://localhost:8080/webstore/` URL on the browser. So let's configure the Dispatcher servlet for our project:

1. Create a class called `WebApplicationContextConfig` under the `com.packt.webstore.config` package in the source directory `src/main/java` and add the following code into it:

```
package com.packt.webstore.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation
.DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation
```

```
.EnableWebMvc;
import org.springframework.web.servlet.config.annotation
.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view
.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan("com.packt.webstore")
public class WebApplicationContextConfig extends
WebMvcConfigurerAdapter {

    @Override
    public void configureDefaultServletHandling
(DefaultServletHandlerConfigurer configurator) {
        configurator.enable();
    }

    @Bean
    public InternalResourceViewResolver
getInternalResourceViewResolver() {
        InternalResourceViewResolver resolver = new
InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/jsp/");
        resolver.setSuffix(".jsp");

        return resolver;
    }
}
```

2. Create a class called `DispatcherServletInitializer` under the `com.packt.webstore.config` package in the source directory `src/main/java` and add the following code into it:

```
package com.packt.webstore.config;

import org.springframework.web.servlet.support
.AbstractAnnotationConfigDispatcherServletInitializer;

public class DispatcherServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }
}
```

```
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[] {
        WebApplicationContextConfig.class };
}

@Override
protected String[] getServletMappings() {
    return new String[] { "/" };
}
}
```

What just happened?

If you know about servlet programming, you might be quite familiar with the servlet configuration and `web.xml`, and in that case, you can consider the `DispatcherServletInitializer` class as similar to `web.xml`. In step 2, we configured a servlet of type `DispatcherServlet` by extending the `AbstractAnnotationConfigDispatcherServletInitializer` class, which is more or less similar to any other normal servlet configuration. The only difference is that we have not instantiated the `DispatcherServlet` class for that configuration. Instead, the servlet class (`org.springframework.web.servlet.DispatcherServlet`) is provided by the Spring MVC framework and we initialized it using the `AbstractAnnotationConfigDispatcherServletInitializer` class.

After this step, our configured `DispatcherServlet` will be ready to handle any requests that come to our application on runtime and will dispatch the request to the correct controller's method.

However, `DispatcherServlet` should know how to access the controller instances and view files that are located in our project, and only then can it properly dispatch the request to the correct controllers. So we have to give some hint to `DispatcherServlet` to locate the controller instances and view files.

This is what we configured within the `getServletConfigClasses` method of the `DispatcherServletInitializer` class. By overriding the `getServletConfigClasses` method, we are telling `DispatcherServlet` about our controller classes and view files. And in step 1, through the `WebApplicationContextConfig` class file, we configured the `InternalResourceViewResolver` and other default configurations.

Don't worry if you are not able to understand each and every configuration in the `WebApplicationContextConfig` and `DispatcherServletInitializer` classes; we will take a look deep into these configuration files in next chapter. As of now, just remember that this is a one-time configuration that is needed to run our project successfully.

Deploying our project

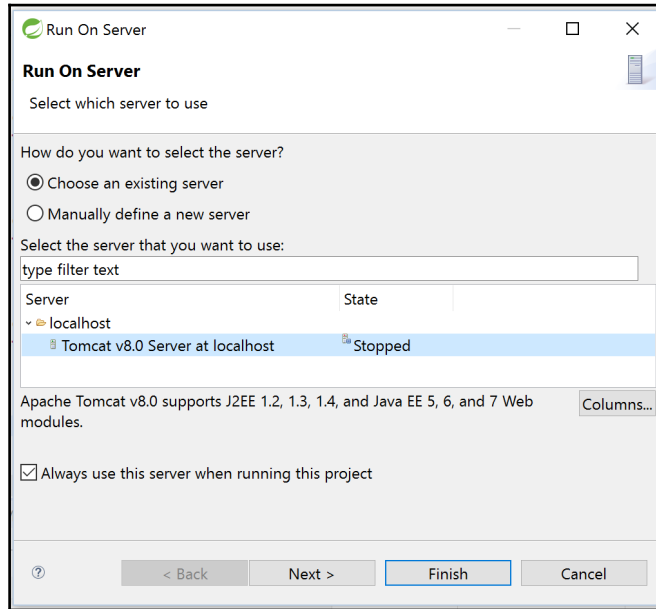
We successfully created the project in the last section, so you might be curious to know what would happen if we run our project now. As our project is a web project, we need a web server to run it.

Time for action – running the project

As we have already configured the Tomcat web server in our STS, let's use Tomcat to deploy and run our project:

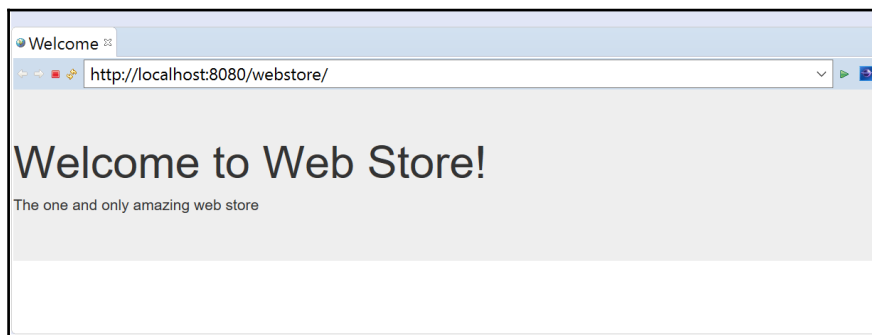
1. Right-click on your project from **Package Explorer** and navigate to **Run As | Run on Server**.
2. A server selection window will appear with all the available servers listed; just select the server that we have configured, **Tomcat v8.0**.

3. At the bottom of the window, you can see a checkbox with the caption that says **Always use this server when running this project**; select this checkbox and enter the **Finish** button, as shown in the following screenshot:



Configuring the default server for a Spring MVC project

4. Now you will see a web page that will show you a welcome message.



Showing the welcome page of the web store

Summary

In this chapter, you saw how to install all the prerequisites that are needed to get started and run your first Spring MVC application, for example, installing JDK, the Maven build tool, the Tomcat servlet container, and STS IDE.

You also learned how to perform various configurations in our STS IDE for Maven and Tomcat, created your first Spring MVC project, and added all Spring-related dependent jars through the Maven configuration.

We had a quick hands-on experience of developing a welcome page for our `webstore` application. During that course, you learned how to put values into a model and how to retrieve these values from the model.

Whatever we have seen so far is just a glimpse of Spring MVC, but there is much more to uncover, for example, how the model, view and controllers are connected to each other and how the request flow occurs. We are going to explore these topics in the next chapter, so see you there.

2

Spring MVC Architecture – Architecting Your Web Store

What we saw in the first chapter was nothing but a glimpse of Spring MVC. Our total focus was just to get a Spring MVC application running. Now it's time for us to deep dive into the Spring MVC architecture.

By the end of this chapter, you will have a clear understanding of:

- The Dispatcher servlet and request mapping
- Web application context and configuration
- Spring MVC request flow and Web MVC
- A typical Spring web application architecture

Dispatcher servlet

In the first chapter, we provided a little introduction to the Dispatcher servlet and you saw how to configure a Dispatcher servlet using the `DispatcherServletInitializer` class. You learned that every web request first comes to the Dispatcher servlet. The Dispatcher servlet is the thing that decides which controller method the web request should be dispatched to. In the previous chapter, we created a welcome page that will be shown whenever we enter the URL `http://localhost:8080/webstore/` in the browser. Mapping a URL to the appropriate controller method is the primary duty of the Dispatcher servlet.

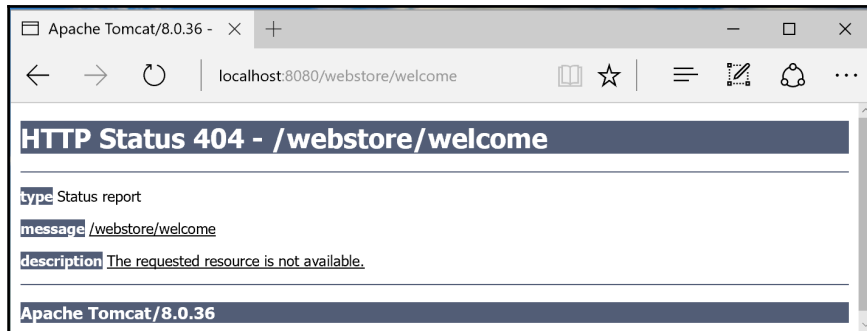
So the Dispatcher servlet reads the web request URL and finds the appropriate controller method that can serve that web request and invokes it. This process of mapping a web request onto a specific controller method is called request mapping. And the Dispatcher servlet is able to do this with the help of the `@RequestMapping` (`org.springframework.web.bind.annotation.RequestMapping`) annotation.

Time for action – examining request mapping

Let's observe what will happen when we change the `value` attribute of the `@RequestMapping` annotation.

1. Open your STS and run your `webstore` project; just right-click on your project and choose **Run As | Run on Server**. Now you will be able to see the same welcome message in the browser.
2. Now go to the address bar of the browser and enter the following URL `http://localhost:8080/webstore/welcome`.
3. You will see the **HTTP Status 404** error page in the browser, and you will also see the following warning in the console:

```
WARNING: No mapping found for HTTP request with URI
[/webstore/welcome] in DispatcherServlet with name
'DefaultServlet'
```



Error showing no mapping found message

4. Now open your `HomeController` class and change the `@RequestMapping` annotation's `value` attribute to `/welcome` and save it. Basically, your new request mapping annotation will look like as follows: `@RequestMapping("/welcome")`.

5. Again, run your application and enter the same URL that you entered in step 2. Now you should be able to see the same welcome message again in the browser without any request mapping error.

What just happened?

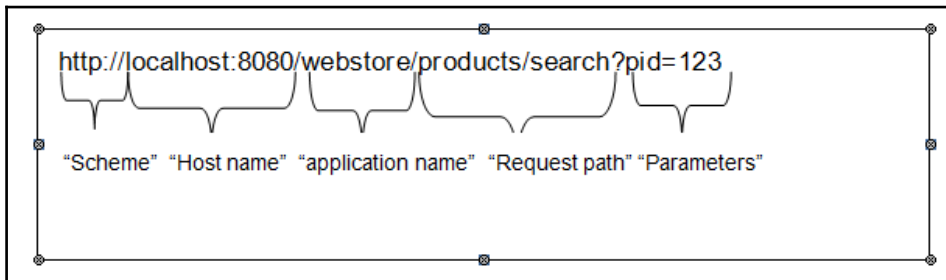
After starting our application, when we enter the URL

`http://localhost:8080/webstore/welcome` in the browser; the Dispatcher servlet (`org.springframework.web.servlet.DispatcherServlet`) immediately tried to find a matching controller's method for the request path `/welcome`.

In a Spring MVC application, a URL can be logically divided into five parts; see the figure that is present after this tip. The `@RequestMapping` annotation only matches against the URL request path; it will omit the scheme, host name, application name, and so on. Here the application name is just a context name where the application is deployed—it is totally under the control of how we configure the web server.



The `@RequestMapping` annotation has one more attribute called `method` to further narrow down the mapping based on HTTP request method types (GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE). If we do not specify the `method` attribute in the `@RequestMapping` annotation, the default `method` would be GET. You will see more about the `method` attribute of the `@RequestMapping` annotation in Chapter 4, *Working with Spring Tag Libraries* under *Time for action – serving and processing forms*.



The logical parts of a typical Spring MVC application URL

But since we don't have a corresponding request mapping for the given URL path `/welcome`, we are getting the **HTTP status 404** error in the browser and the following error log in the console:

```
WARNING: No mapping found for HTTP request with URI [/webstore/welcome] in
DispatcherServlet with name 'DefaultServlet'
```

From the error log, we can clearly understand that there is no request mapping for the URL path `/webstore/welcome`. So we are trying to map this URL path to the existing controller's method; that's why in step 4 we only put the request path value `/welcome` in the `@RequestMapping` annotation as the value attribute. Now everything works perfectly fine.

Pop quiz – request mapping

Suppose I have a Spring MVC application for library management called *BookPedia* and I want to map a web request URL

`http://localhost:8080/BookPedia/category/fiction` to a controller's method—how would you form the `@RequestMapping` annotation?

1. `@RequestMapping("/fiction")`
2. `@RequestMapping("/category/fiction")`
3. `@RequestMapping("/BookPedia/category/fiction")`

What is the request path in the following URL `http://localhost:8080/webstore/?`

1. `webstore/`
2. `/`
3. `8080/webstore/`

Understanding the Dispatcher servlet configuration

Now we've got a basic idea of how request mapping works. I also mentioned that every web request first comes to the Dispatcher servlet, but the question is how does the Dispatcher servlet know it should handle every incoming request? The answer is we explicitly instructed it to do so through the `getServletMappings` method of the `DispatcherServletInitializer` class. Yes, when we return the string array containing

only the “/” character, it indicates the `DispatcherServlet` configuration as the default servlet of the application. So every incoming request will be handled by `DispatcherServlet`.

Time for action – examining the servlet mapping

Let's observe what will happen when we change the return value of the `getServletMappings` method.

1. Open `DispatcherServletInitializer` and change the return value of the `getServletMappings` method as `return new String[] { "/app/*"}`; basically your `getServletMappings` method should look like the following after your change:

```
@Override
protected String[] getServletMappings() {
    return new String[] { "/app/*" };
}
```

2. Run your application by right-clicking on your project and choose **Run As | Run on Server**.
3. Go to the address bar of the browser and enter the following URL `http://localhost:8080/webstore/welcome`. You will see the **HTTP Status 404** error page in the browser.
4. Again go to the address bar of the browser and enter the following URL `http://localhost:8080/webstore/app/welcome` and you will be able to see the same welcome message in the browser.
5. Now revert the return value of the `getServletMappings` method to its original value. Basically, your `getServletMappings` method should look as follows after your change:

```
@Override
protected String[] getServletMappings() {
    return new String[] { "/" };
}
```

What just happened?

As I have already mentioned, we can consider the `DispatcherServletInitializer` class as equivalent to `web.xml` as it extends from the `AbstractAnnotationConfigDispatcherServletInitializer`. If you look close enough, we are overriding three important methods in `DispatcherServletInitializer`, namely:

- `getRootConfigClasses`: This specifies the configuration classes for the root application context
- `getServletConfigClasses`: This specifies the configuration classes for the Dispatcher servlet application context
- `getServletMappings`: This specifies the servlet mappings for `DispatcherServlet`



Typically, the context loaded using the `getRootConfigClasses` method is the *root* context, which belongs to the whole application, while the one initialized using the `getServletConfigClasses` method is actually specific to that Dispatcher servlet. Technically, you can have multiple Dispatcher servlets in an application and so multiple such contexts, each specific for the respective Dispatcher servlet but with the same root context.

In step 1, when we changed the return value of the `getServletMappings` method, we instructed `DispatcherServlet` to handle only the web requests that start with a prefix text of `/app/`. That's why we entered the URL

`http://localhost:8080/webstore/welcome`. We saw the **HTTP Status 404** error page in the browser, since the URL request path didn't start with the `/app/` prefix. But when we tried the URL `http://localhost:8080/webstore/app/welcome`, we were able to see the same welcome message in the browser as the URL started with the `/app/` prefix.

Servlet mapping versus request mapping

The servlet mapping specifies which web container of the Java servlet should be invoked for a given URL. It maps the URL patterns to servlets. When there is a request from a client, the servlet container decides which servlet it should forward the request to based on the servlet mapping. In our case, we mapped all incoming requests to `DispatcherServlet`.

In contrast, request mapping guides the `DispatcherServlet` which controller method it needs to invoke as a response to the request based on the request path. In our case, we mapped the `/welcome` request path to the `welcome` method of the `HomeController` class.

Pop quiz – servlet mapping

Considering the following servlet mapping, identify the possible matching URLs:

```
@Override
protected String[] getServletMappings() {
    return new String[] { "*.do" };
}
```

1. `http://localhost:8080/webstore/welcome`
2. `http://localhost:8080/webstore/do/welcome`
3. `http://localhost:8080/webstore/welcome.do`
4. `http://localhost:8080/webstore/welcome/do`

Considering the following servlet mapping, identify the possible matching URLs:

```
@Override
protected String[] getServletMappings() {
    return new String[] { "/" };
}
```

1. `http://localhost:8080/webstore/welcome`
2. `http://localhost:8080/webstore/products`
3. `http://localhost:8080/webstore/products/computers`
4. All the above

Web application context

In a Spring-based application, our application objects will live within an object container. This container will create objects and associations between objects and manage their complete lifecycle. These container objects are called Spring managed beans (or simply beans) and the container is called application context in the Spring world.

Spring's container uses **dependency injection (DI)** to manage the beans that make up an application. An application context (`org.springframework.context.ApplicationContext`) creates beans, associates beans together based on bean configuration, and dispenses beans upon request. A bean configuration can be defined via an XML file, annotation, or even via Java configuration classes. We are going to use annotation and Java configurations in our chapters.

A **web application context** is an extension of the application context, and is designed to work with the standard servlet context (`javax.servlet.ServletContext`). The web application context typically contains front-end related beans such as views and view resolvers, and so on. In the first chapter, we simply created a class called `WebApplicationContextConfig`, which is a bean configuration for our web application.

We learned that `WebApplicationContextConfig` is nothing but a Java-based bean configuration file for our web application context, where we can define the beans to be used in our application. Usually, we define beans using the `@Bean` annotation. In order to run a Spring MVC application successfully, Spring needs at least a bean that implements the `org.springframework.web.servlet.ViewResolver` interface. One such bean we defined in our web application context is `InternalResourceViewResolver`.

View resolvers

A view resolver helps the Dispatcher servlet to identify the views that have to be rendered as a response to a specific web request. Spring MVC provides various view resolver implementations to identify views and `InternalResourceViewResolver` is one such implementation:

```
@Bean
public InternalResourceViewResolver getInternalResourceViewResolver() {
    InternalResourceViewResolver resolver = new
InternalResourceViewResolver();
    resolver.setViewClass(JstlView.class);
    resolver.setPrefix("/WEB-INF/jsp/");
    resolver.setSuffix(".jsp");

    return resolver;
}
```

Through the above bean definition is in the web application context configuration (`WebApplicationContextConfig`), we are instructing Spring MVC to create a bean for the class `InternalResourceViewResolver` (`org.springframework.web.servlet.view.InternalResourceViewResolver`). We will see more about the view resolver in Chapter 5, *Working with View Resolver*.

Time for action – understanding web application context

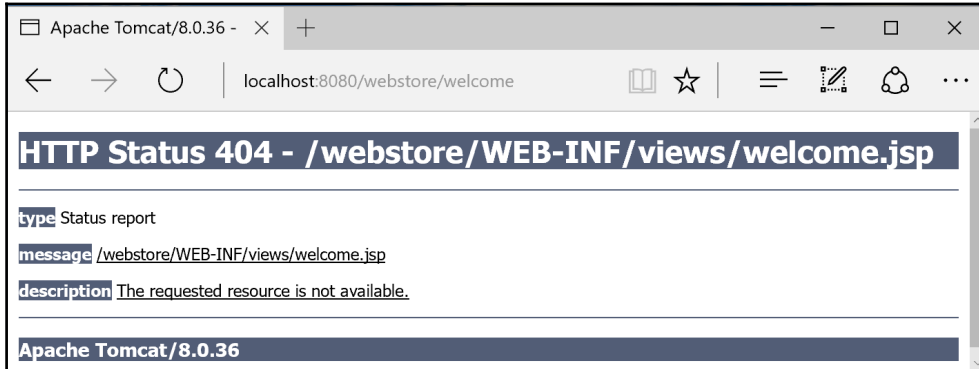
Okay we have seen enough introductions about web application context now, let's tweak the `InternalResourceViewResolver` bean from the web application context configuration (`WebApplicationContextConfig.java`) a little bit and observe the effect.

1. Open `WebApplicationContextConfig.java` and set the prefix as `/WEB-INF/views/` for the `InternalResourceViewResolver` bean. Basically your `InternalResourceViewResolver` bean definition should look as follows after your change:

```
@Bean
public InternalResourceViewResolver
getInternalResourceViewResolver() {
    InternalResourceViewResolver resolver = new
    InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");

        return resolver;
}
```

2. Run your application, then go to the address bar of the browser and enter the following URL: `http://localhost:8080/webstore/welcome`. You will see the **HTTP Status 404** error page in the browser:



The logical parts of a typical Spring MVC application URL

3. To fix this error, rename the folder `/webstore/src/main/webapp/WEB-INF/jsp` to `/webstore/src/main/webapp/WEB-INF/views`.
4. Now run your application and enter the URL `http://localhost:8080/webstore/welcome`; you will see the welcome message again.

What just happened?

After changing the `prefix` property value of the `InternalResourceViewResolver` bean, when we entered the URL `http://localhost:8080/webstore/welcome` in the browser, we got a **HTTP status 404** error. The **HTTP status 404** error means that the server could not find the web page that you asked for. Okay if that is the case, which web page did we ask for?

As a matter of fact, we didn't ask for any web page to the server directly; instead the Dispatcher servlet asked for a particular web page to the server. And what we already learned is that the Dispatcher servlet will invoke a method in any of our controller beans that can serve this web request. In our case, that method is nothing but the `welcome` method of our `HomeController` class, because that is the only request mapping method that can match the request path of given URL

`http://localhost:8080/webstore/welcome` in its `@RequestMapping` annotation.

Now I want you to observe three things:

1. The `prefix` property value of the `InternalResourceViewResolver` bean definition in `WebApplicationContextConfig.java`:

```
/WEB-INF/views/
```

2. The return value of the `welcome` method from the `HomeController` class:

```
welcome
```

3. Finally, the `suffix` property value of the `InternalResourceViewResolver` bean:

```
.jsp
```

If you combine these three values together, you will get a web page request URL as `/WEB-INF/views/welcome.jsp`. Now notice the error message in the figure after step 2, which is showing a **HTTP status 404** error for the same web page URL `/WEB-INF/views/welcome.jsp` under the application name `/webstore/`.

So the conclusion is that `InternalResourceViewResolver` resolves the actual view's file path by prepending the configured `prefix` and appending the `suffix` value with the view name. The view name is the value usually returned by controller's method. So the controller's method doesn't return the path of the actual view file-it just returns the logical view name. It is the job of `InternalResourceViewResolver` to form the URL of the actual view file correctly.

Okay fine, but who is going to use this final formed URL? The answer is the Dispatcher servlet. Yes, after getting the final formed URL of a view file from the view resolver, the Dispatcher servlet will try to get the view file from the server. During that time, if the formed URL is found to be wrong, then you will get a **HTTP status 404** error.

Usually after invoking the controller's method, the Dispatcher servlet will wait to get the logical view name from the controller's method. Once the Dispatcher servlet gets the logical view name, it will give that to the view resolver (`InternalResourceViewResolver`) to get the URL path of the actual view file. Once the view resolver returns the URL path to the Dispatcher servlet, the rendered view file is served to the client browser as a web page by the Dispatcher servlet.

Okay fine, but why did we get the error in step 2? Since we changed the `prefix` property of `InternalResourceViewResolver` in step 2, the URL path value returned from `InternalResourceViewResolver` would become `/WEB-INF/views/welcome.jsp` in step 3, which is an invalid path value (there is no directory called `views` under `WEB-INF`). That's why we renamed the directory `jsp` to `views` in step 3 to align it with the path generated by `InternalResourceViewResolver`, so everything works fine again.

Understanding the web application context configuration

The web application context configuration file (`WebApplicationContextConfig.java`) is nothing but a simple Java-based Spring bean configuration class. Spring will create beans (objects) for every bean definition mentioned in this class during the boot up of our application. If you open this web application context configuration file, you will find the following annotations on top of the class definition:

- `@Configuration`: This indicates that a class declares one or more `@Bean` methods
- `@EnableWebMvc`: Adding this annotation to an `@Configuration` class imports some special Spring MVC configuration
- `@ComponentScan`: This specifies the base packages to scan for annotated components (beans)

The first annotation `@Configuration` indicates that this class declares one or more `@Bean` methods. If you remember, in the last section, I explained how we created a bean definition for `InternalResourceViewResolver`.

The second annotation is `@EnableWebMvc`. With this annotation, we are telling Spring MVC to configure the `DefaultAnnotationHandlerMapping`, `AnnotationMethodHandlerAdapter` and `ExceptionHandlerExceptionResolver` beans. These beans are required for Spring MVC to dispatch requests to the controllers.

Actually, `@EnableWebMvc` does many things behind the screen. It also enables support for various convenient annotations such as `@NumberFormat`, `@DateTimeFormat` to format the form bean's fields during form binding, and similarly the `@Valid` annotation to validate the controller method's parameters. It even supports Java objects being converted to/from XML or JSON via the `@RequestBody` and `@ResponseBody` annotation in the `@RequestMapping` or `@ExceptionHandler` methods during form binding. We will see the usage of these

annotations in later chapters. As for now, just remember that the `@EnableWebMvc` annotation is needed to enable annotations such as `@controller` and `@RequestMapping` and so on.

Now the third annotation `@ComponentScan`-what is the purpose of this annotation? You need a little bit of background information to understand the purpose of the `@ComponentScan` annotation. The `@Controller` annotation indicates that a particular class serves the role of a controller. You have already learned that the Dispatcher servlet searches such annotated classes for mapped methods (`@RequestMapping` annotated methods) to serve a web request. In order to make the controller available for searching, we must create a bean for that controller in our web application context.

We can create beans for controllers explicitly via the bean configuration (using the `@Bean` annotation; you can see how we created a bean for the `InternalResourceViewResolver` class using the `@Bean` annotation for reference), or we can hand over that task to Spring via an auto-detection mechanism. To enable auto-detection of the `@Controller` annotated classes, we must add component scanning to our configuration using the `@ComponentScan` annotation. Now you understand the purpose of the `@ComponentScan` annotation.

Spring will create beans (objects) for every `@Controller` class at runtime. The Dispatcher servlet will search for the correct request mapping method in every `@Controller` bean based on the `@RequestMapping` annotation to serve a web request. The `base-package` property of a `@ComponentScan` annotation indicates under which package Spring should search for controller classes to create beans:

```
@ComponentScan("com.packt.webstore")
```

This line instructs Spring to search for controller classes in the `com.packt.webstore` package and its sub-packages.



The `@ComponentScan` annotation not only recognizes controller classes, it will also recognize other stereotypes such as services and repositories classes as well. We will explore services and repositories later.

Pop quiz – web application context configuration

In order to identify a class as a controller by Spring, what needs to be done?

1. That particular class should have an `@Controller` annotation.
2. The `@EnableWebMvc` annotation and `@ComponentScan` annotation should be specified in the web application context configuration file.

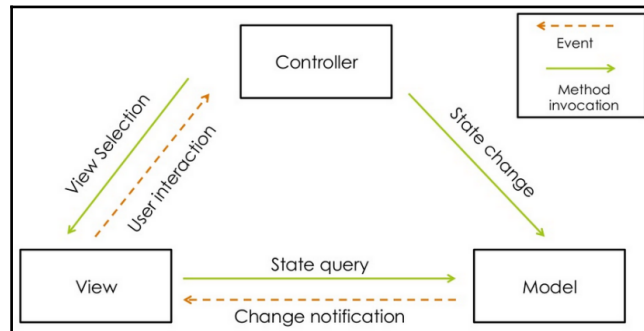
3. That particular class should be put up in a package or in a sub-package that has been specified as a value in the `@ComponentScan` annotation.
4. All of the above.

Model View Controller

So far, we have looked at lots of concepts such as the Dispatcher servlet, request mapping, controllers, and the view resolver, but it would be good to see the overall picture of the Spring MVC request flow so that we can understand each component's responsibilities. But before that, you need a basic understanding of the **Model View Controller (MVC)** concept. Every enterprise level application's Presentation layer can be logically divided into three major parts:

- The part that manages the data (**Model**)
- The part that creates the user interface and screens (**View**)
- The part that handles interactions between the user, the user interface, and the data (**Controller**)

The following diagram should help you to understand the event flow and command flow within an MVC pattern.



The classic MVC pattern

Whenever a user interacts with the view by clicking on a link or button, or something similar, the view issues an event notification to the controller and the controller issues a command notification to the model to update the data. Similarly, whenever the data in the model is updated or changed, a change notification event is issued to the view by the model, and in response the view issues a state query command to the model to get the latest

data from the model. Here, the model and view can directly interact. This pattern is called the classic MVC. But what Spring MVC employs is something called the Web MVC pattern because of the limitation in the HTTP protocol.

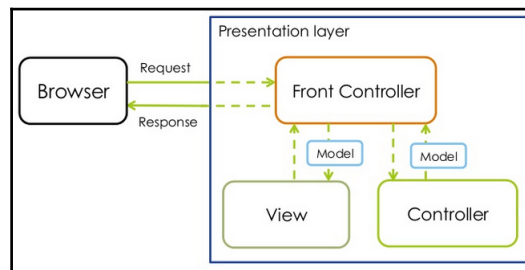


Web applications rely on the HTTP protocol, which is a stateless pull protocol. This means no request implies no reply every time we need to make a request to the application to know the state of the application. The MVC design pattern requires a push protocol for the views to be notified by the model. So the Web MVC controller takes more responsibility for state changing, state querying and change notification.

In Web MVC, every interaction between the model and view are done via controllers only. So the controller acts as a bridge between the model and the view. There is no direct interaction between the model and the view as in the classic MVC.

Overview of the Spring MVC request flow

The main entry point for a web request in a Spring MVC application is via the Dispatcher servlet. The Dispatcher servlet acts as a front controller and dispatches the requests to the other controller. The front controller's main duty is to find the appropriate right controller to hand over the request for further processing. The following diagram shows an overview of the request flow in a Spring MVC application:



The Spring MVC request flow

Okay, let's review the Spring MVC request flow in short:

1. When we enter a URL in the browser, the request comes to the Dispatcher servlet. The Dispatcher servlet acts as a centralized entry point to the web application.
2. The Dispatcher servlet determines a suitable controller that is capable of handling the request and dispatching that request to the controller.

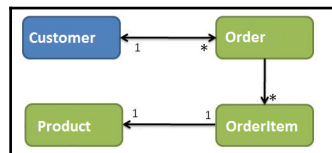
3. The controller method updates the objects in the model and returns the logical view name and updated model to the Dispatcher servlet.
4. The Dispatcher servlet consults with view resolver to determine which view to render and passes the model's data to that view.
5. View furnishes the dynamic values in the web page using the model's data and renders the final web page and returns that web page to the Dispatcher servlet.
6. At the end, the Dispatcher servlet returns the final rendered page as a response to the browser.

The web application architecture

Now you understand the overall request flow and the responsibility of each component in a typical Spring MVC application, but that is not enough for you to build an online web store application. We also need to know the best practices to develop an enterprise-level web application. One of the best practices in a typical web application is to organize source codes into layers, which will improve reusability and loose coupling. A typical web application would normally have four layers, namely presentation, domain, services, and persistence. So far, what we have seen like the Dispatcher servlet, controllers, view resolvers, and similar, are considered to be part of the Presentation layer's components. Now you need to understand the remaining layers and components one by one.

The Domain layer

So let's start with the Domain layer. The Domain layer typically consists of a domain model. So what is a domain model? A domain model is a representation of the data storage types required by the business logic. It describes the various domain objects (entities), their attributes, roles, and relationships, plus the constraints that govern the problem domain. You can look at the following order processing domain model diagram to get a quick idea about the domain model.



A sample domain model

Each block in the previous diagram represents a business entity and the lines represent the associations between the entities. Based on this domain model diagram, you should understand that in an order processing domain, a `Customer` can have many `Order` and each `Order` can have many `OrderItem` and each `OrderItem` represents a single `Product`.

During actual coding and development this domain model, will be converted into corresponding domain objects and associations by a developer. A domain object is a logical container of purely domain information. Since we are going to build an online web store application, in our domain the primary domain object might be a product. So let's start with the domain object to represent a product.

Time for action – creating a domain object

So far in our web store, we have displayed only a welcome message. It is time for us to show our first product on our web page. Let's do this by creating a domain object to represent the product information.

1. Create a class called `Product` under the `com.packt.webstore.domain` package in the `src/main/java` source folder and add the following code into it:

```
package com.packt.webstore.domain;

import java.io.Serializable;
import java.math.BigDecimal;

public class Product implements Serializable {
    private static final long serialVersionUID =
3678107792576131001L;

    private String productId;
    private String name;
    private BigDecimal unitPrice;
    private String description;
    private String manufacturer;
    private String category;
    private long unitsInStock;
    private long unitsInOrder;
    private boolean discontinued;
    private String condition;

    public Product() {
        super();
    }

    public Product(String productId, String name, BigDecimal
```

```
        unitPrice) {
            this.productId = productId;
            this.name = name;
            this.unitPrice = unitPrice;
        }

        // add setters and getters for all the fields here

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Product other = (Product) obj;
            if (productId == null) {
                if (other.productId != null)
                    return false;
            } else if (!productId.equals(other.productId))
                return false;
            return true;
        }

        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result
                + ((productId == null) ? 0 :
                    productId.hashCode());
            return result;
        }
    }
}
```

2. Add setters and getters for all the fields as well as for the previous class. I have omitted it to make the code compact, but it is really needed, so please do add setters and getters for all the fields except `serialVersionUID` field.
3. Now create one more controller class called `ProductController` under the `com.packt.webstore.controller` package in the `src/main/java` source folder. And add the following code into it:

```
package com.packt.webstore.controller;

import java.math.BigDecimal;
import org.springframework.stereotype.Controller;
```

```
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import com.packt.webstore.domain.Product;

@Controller
public class ProductController {

    @RequestMapping("/products")
    public String list(Model model) {
        Product iphone = new Product("P1234", "iPhone 6s", new
BigDecimal(500));
        iphone.setDescription("Apple iPhone 6s smartphone
with 4.00-inch 640x1136 display and 8-megapixel rear
camera");
        iphone.setCategory("Smartphone");
        iphone.setManufacturer("Apple");
        iphone.setUnitsInStock(1000);
        model.addAttribute("product", iphone);
        return "products";
    }
}
```

4. Finally, add one more JSP view file called `products.jsp` under the `src/main/webapp/WEB-INF/views/` directory, add the following code snippets into it, and save it:

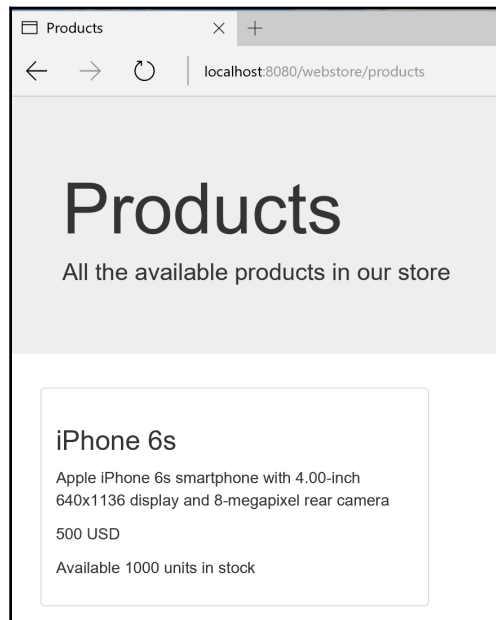
```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css">
<title>Products</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Products</h1>
                <p>All the available products in our store</p>
            </div>
        </div>
    </section>

    <section class="container">
```

```
<div class="row">
  <div class="col-sm-6 col-md-3" style="padding-
bottom: 15px">
    <div class="thumbnail">
      <div class="caption">
        <h3>${product.name}</h3>
        <p>${product.description}</p>
        <p>${product.unitPrice} USD</p>
        <p>Available ${product.unitsInStock} units in stock</p>
      </div>
    </div>
  </div>
</div>
</section>
</body>
</html>
```

5. Now run your application and enter the URL `http://localhost:8080/webstore/products` You should be able to see a web page showing product information as shown in the following figure:



Products page showing product information

What just happened?

Our aim is to show the details of a product in our web page. In order to do that, first we need a domain object to hold the details of a product. That's what we did in step 1; we just created a class called `Product` (`Product.java`) to store information about the product such as the name, description, price, and more.

As you have already learned in the *Overview of Spring MVC request flow* section, to show any dynamic data in a web page, prior to doing so we need to put that data in a model, then only the view can read that data from the model and will render it in the web page. So to put product information in a model, we just created one more controller called `ProductController` (`ProductController.java`) in step 3.

In `ProductController`, we just have a single method called `list` whose responsibility it is to create a product domain object to hold the information about Apple's iPhone 5s and add that object to the model. And finally, we return the view name as `products`. That's what we were doing in the following lines of the `list` method of `ProductController`:

```
model.addAttribute("product", iphone);
return "products";
```

Since we configured `InternalResourceViewResolver` as our view resolver in the web application context configuration, during the process of resolving the view file for the given view name (in our case the view name is `products`), the view resolver will try to look for a file called `products.jsp` under `/WEB-INF/views/`. That's why we created `products.jsp` in step 4. If you skipped step 4, you will get a **HTTP status 404** error while running the project.

For a better visual experience, `products.jsp` contains lots of `<div>` tags with Bootstrap CSS styles applied (Bootstrap is an open source CSS framework). So don't think that `products.jsp` is very complex; as a matter of fact it is very simple-you don't need to bother about the `<div>` tags, as those are present just to get an appealing look. You only need to observe the following four tags carefully in `products.jsp` to understand the data retrieval from the model:

```
<h3>${product.name}</h3>
<p>${product.description}</p>
<p>${product.unitPrice} USD</p>
<p>Available ${product.unitsInStock} units in stock</p>
```

Look carefully at the expression `${product.unitPrice}`. The `product` text in the expression is nothing but the name of the key. We used this key to store the `iphone` domain object in the model; (remember this line `model.addAttribute("product", iphone)` from `ProductController`) and the `unitPrice` text is nothing but one of the fields from the `Product` domain class (`Product.java`). Similarly we are showing some important fields of the `product` domain class in the `products.jsp` file.



When I say that `price` is the field name, I am actually making an assumption here that you have followed the standard Java bean naming conventions for the getters and setters of your domain class. When Spring evaluates the expression `${product.unitPrice}`, it is actually trying to call the getter method of the field to get the value, so it would expect the `getUnitPrice()` method to be in the `Product.java` file.

After finishing step 4, if we run our application and enter the URL `http://localhost:8080/webstore/products`, we are able to see a web page showing product information as shown in the screenshot after step 5.

So we created a domain class to hold information about a product, created a single product object in the controller and added it to the model, and finally showed that product's information in the view.

The Persistence layer

Since we had a single product, we just instantiated it in the controller itself and successfully showed the product information on our web page. But a typical web store would contain thousands of products, so all the product information for them would usually be stored in a database. This means we need to make our `ProductController` smart enough to load all the product information from the database into the model. But if we write all the data retrieval logic to retrieve the product information from the database in the `ProductController` itself, our `ProductController` will blow up into a big chunk of files. And logically speaking, data retrieval is not the duty of the controller because the controller is a Presentation layer component. And moreover, we want to organize the data retrieval code into a separate layer, so that we can reuse that logic as much as possible from the other controllers and layers.

So how do we retrieve data *from a database* in a Spring MVC way? Here comes the concept of the Persistence layer. A Persistence layer usually contains repository objects to access domain objects. A repository object sends queries to the data source for the data, then it maps the data from the data source to a domain object, and finally it persists the changes in

the domain object to the data source. So typically, a repository object is responsible for CRUD (Create, Read, Update, and Delete) operations on domain objects. And the `@Repository (org.springframework.stereotype.Repository)` annotation is an annotation that marks the specific class as a repository. The `@Repository` annotation also indicates that `SQLExceptions` thrown from the repository object's methods should be translated into Spring's specific `org.springframework.dao.DataAccessExceptions`. Let's create a repository layer for our application.

Time for action – creating a repository object

Let's create a repository class to access our `Product` domain objects.

1. Open `pom.xml` to add a dependency to **spring-jdbc**. In **Group Id** enter `org.springframework`, in **Artifact Id** enter `spring-jdbc`, and in **Version** enter `4.3.0.RELEASE`. Select **Scope** as **compile** and then click on the **OK** button.
2. Similarly, add the dependency for **HyperSQL DB** by clicking on the same **Add** button. This time, enter `org.hsqldb` for **Group Id**, `hsqldb` for **Artifact Id**, `2.3.2` for **Version**, select **Scope** as **compile**, and save `pom.xml`.
3. Create a class called `RootApplicationContextConfig` under the `com.packt.webstore.config` package in the `src/main/java` source folder and add the following code to it:

```
package com.packt.webstore.config;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation
    .ComponentScan;
import org.springframework.context.annotation
    .Configuration;
import org.springframework.jdbc.core.namedparam
    .NamedParameterJdbcTemplate;
import org.springframework.jdbc.datasource.embedded
    .EmbeddedDatabase;
import org.springframework.jdbc.datasource.embedded
    .EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded
    .EmbeddedDatabaseType;

@Configuration
@ComponentScan("com.packt.webstore")
public class RootApplicationContextConfig {
```

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new
    EmbeddedDatabaseBuilder();
    EmbeddedDatabase db = builder
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("db/sql/create-table.sql")
        .addScript("db/sql/insert-data.sql")
        .build();
    return db;
}
@Bean
public NamedParameterJdbcTemplate getJdbcTemplate() {
    return new NamedParameterJdbcTemplate(dataSource());
}
}
```

4. Create a folder structure called `db/sql/` under the `src/main/resources` source folder and create a file called `create-table.sql` in it. Add the following SQL script to it:

```
DROP TABLE PRODUCTS IF EXISTS;

CREATE TABLE PRODUCTS (
    ID VARCHAR(25) PRIMARY KEY,
    NAME VARCHAR(50),
    DESCRIPTION VARCHAR(250),
    UNIT_PRICE DECIMAL,
    MANUFACTURER VARCHAR(50),
    CATEGORY VARCHAR(50),
    CONDITION VARCHAR(50),
    UNITS_IN_STOCK BIGINT,
    UNITS_IN_ORDER BIGINT,
    DISCONTINUED BOOLEAN
);
```

5. Similarly create one more SQL script file called `insert-data.sql` under the `db/sql` folder and add the following script to it:

```
INSERT INTO PRODUCTS VALUES ('P1234', 'iPhone 6s', 'Apple
iPhone 6s smartphone with 4.00-inch 640x1136 display and 8-
megapixel rear
camera', '500', 'Apple', 'Smartphone', 'New', 450, 0, false);

INSERT INTO PRODUCTS VALUES ('P1235', 'Dell Inspiron',
'Dell Inspiron 14-inch Laptop (Black) with 3rd Generation      Intel
Core processors',
```

```
700, 'Dell', 'Laptop', 'New', 1000, 0, false);

INSERT INTO PRODUCTS VALUES ('P1236', 'Nexus 7', 'Google
Nexus 7 is the lightest 7 inch tablet With a quad-core           Qualcomm
Snapdragon™ S4 Pro processor',
300, 'Google', 'Tablet', 'New', 1000, 0, false);
```

6. Open `DispatcherServletInitializer` and change the `getRootConfigClasses` method's return value to return new `Class[] { RootApplicationContextConfig.class }`; Basically, your `getRootConfigClasses` method should look as follows after your change:

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[] { RootApplicationContextConfig.class
};
}
```

7. Create an interface called `ProductRepository` under the `com.packt.webstore.domain.repository` package in the `src/main/java` source folder. And add a single method declaration in the interface as follows:

```
package com.packt.webstore.domain.repository;

import java.util.List;

import com.packt.webstore.domain.Product;

public interface ProductRepository {

    List <Product> getAllProducts();
}
```

8. Create a class called `InMemoryProductRepository` under the `com.packt.webstore.domain.repository.impl` package in the `src/main/java` source folder and add the following code to it:

```
package com.packt.webstore.domain.repository.impl;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation
```

```
.Autowired;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam
.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import com.packt.webstore.domain.Product;
import com.packt.webstore.domain.repository
.ProductRepository;

@Repository
public class InMemoryProductRepository implements
ProductRepository{
    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    public List<Product> getAllProducts() {
        Map<String, Object> params = new HashMap<String,
Object>();
        List<Product> result = jdbcTemplate.query("SELECT *
FROM products", params, new ProductMapper());
        return result;
    }

    private static final class ProductMapper implements
RowMapper<Product> {
        public Product mapRow(ResultSet rs, int rowNum)
throws SQLException {
            Product product = new Product();
            product.setProductId(rs.getString("ID"));
            product.setName(rs.getString("NAME"));
            product.setDescription(rs.getString("DESCRIPTION"));
            product.setUnitPrice(rs.getBigDecimal("UNIT_PRICE"));
            product.setManufacturer(rs.getString("MANUFACTURER"));
            product.setCategory(rs.getString("CATEGORY"));
            product.setCondition(rs.getString("CONDITION"));
            product.setUnitsInStock(rs.getLong("UNITS_IN_STOCK"));
            product.setUnitsInOrder(rs.getLong("UNITS_IN_ORDER"));
            product.setDiscontinued(rs.getBoolean("DISCONTINUED"));
            return product;
        }
    }
}
```

9. Open `ProductController` from the `com.packt.webstore.controller` package in the `src/main/java` source folder. Add a private reference to `ProductRepository` with the `@Autowired` (`org.springframework.beans.factory.annotation.Autowired`) annotation as follows:

```
@Autowired
private ProductRepository productRepository;
```

10. Now alter the body of the `list` method as follows in `ProductController`:

```
@RequestMapping("/products")
public String list(Model model) {
    model.addAttribute("products",
        productRepository.getAllProducts());
    return "products";
}
```

11. Finally, open your `products.jsp` view file from `src/main/webapp/WEB-INF/views/` and remove all the existing code and replace it with the following code snippet:

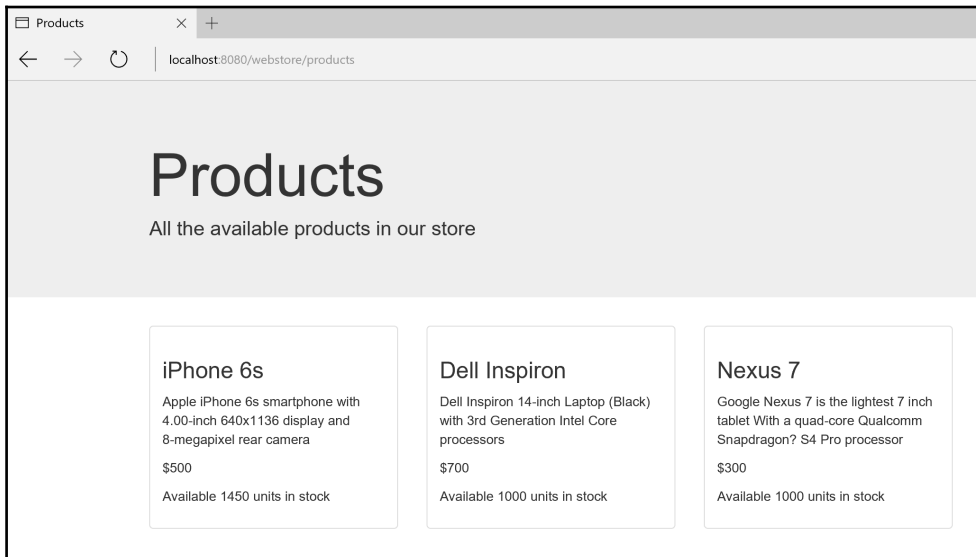
```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Products</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Products</h1>
                <p>All the available products in our store</p>
            </div>
        </div>
    </section>

    <section class="container">
        <div class="row">
```

```
<c:forEach items="${products}" var="product">
  <div class="col-sm-6 col-md-3">
    <div class="thumbnail">
      <div class="caption">
        <h3>${product.name}</h3>
        <p>${product.description}</p>
        <p>${product.unitPrice}</p>
        <p>Available ${product.unitsInStock} units in stock</p>
      </div>
    </div>
  </div>
</c:forEach>
</div>
</section>
</body>
</html>
```

12. Now run your application and enter the URL `http://localhost:8080/webstore/products`. You will see a web page showing product information as shown in the following screenshot:



Products page showing all the products info from the in-memory repository

What just happened?

The most important step in the previous section is step 8, where we created the `InMemoryProductRepository` class. Since we don't want to write all the data retrieval logic inside the `ProductController` itself, we delegated that task to another class called `InMemoryProductRepository`. The `InMemoryProductRepository` class has a single method called `getAllProducts()`, which will return a list of product domain objects.

As the name implies, `InMemoryProductRepository` is trying to communicate with an in-memory database to retrieve all the information relating to the products. We decided to use one of the popular in-memory database implementations called **HyperSQL DB**; that's why we added the dependency for that jar in step 2. And in order to connect and query the HyperSQL database, we decided to use the `spring-jdbc` API. This means we need the `spring-jdbc` jar as well in our project, so we added that dependency in step 1.

Having the required dependency in place, the next logical step is to connect to the database. In order to connect to the database, we need a data source bean in our application context. So in step 3, we created one more bean configuration file called `RootApplicationContextConfig` and added two bean definitions in it. Let's see what they are one by one.

The first bean definition we defined in `RootApplicationContextConfig` is to create a bean for the `javax.sql.DataSource` class:

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    EmbeddedDatabase db = builder
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("db/sql/create-table.sql")
        .addScript("db/sql/insert-data.sql")
        .build();
    return db;
}
```

In this bean definition, we employed `EmbeddedDatabaseBuilder` to build an in-memory database (HyperSQL) with a specified script file to create the initial tables and data to insert. If you watch closely enough, you can see that we are passing two script files to `EmbeddedDatabaseBuilder`:

- `create-table.sql`: This file contains a SQL script to create a product table
- `insert-data.sql`: This file contains a SQL script to insert some initial product records into the product table

So the next step is to create the specified script file in the respective directory so that `EmbeddedDatabaseBuilder` can use that script file in order to initialize the in-memory database. That's what we did in steps 4 and 5.

Okay, using the data source bean, we created and initialized the in-memory database, but in order to communicate with the database we need one more bean called `NamedParameterJdbcTemplate`; that is the second bean we defined in `RootApplicationContextConfig`. If you watch the bean definition for `NamedParameterJdbcTemplate` closely enough, you can see that we have passed the `dataSource()` bean as a constructor parameter to the `NamedParameterJdbcTemplate` bean:

```
@Bean
public NamedParameterJdbcTemplate getJdbcTemplate() {
    return new NamedParameterJdbcTemplate(dataSource());
}
```

Okay, so far so good. We created a bean configuration file (`RootApplicationContextConfig`) and defined two beans in it to initialize and communicate with the in-memory database. But for Spring MVC to actually pick up this bean configuration file and create beans (objects), for those bean definitions we need to hand over this file to Spring MVC during the initialization of our application. That's what we did in step 6 through the `DispatcherServletInitializer` class:

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[] { RootApplicationContextConfig.class };
}
```

As I have already mentioned, we did all the previously specified steps in order to use the `NamedParameterJdbcTemplate` bean in our `InMemoryProductRepository` class to communicate with the in-memory database. You may then be wondering what we did in step 7. In step 1, we are just creating an interface called `ProductRepository`, which defines the expected behavior of a product repository. As of now, the only expected behavior of a `ProductRepository` is to return a list of product domain objects (`getAllProducts`), and our `InMemoryProductRepository` is just an implementation of that interface.

Why do we have an interface and an implementation for the product repository? Remember we are actually creating a Persistence layer for our application. Who is going to use our Persistence layer repository object? Possibly a controller object (in our case `ProductController`) from the Controller layer, so it is not best practice to connect two layers (Controller and Persistence) with a direct reference. Instead we can have an interface

reference in the controller so that in future if we want to, we can easily switch to different implementations of the repository without making any code changes in the controller class.

That's the reason in step 9 that we had the `ProductRepository` reference in our `ProductController` not the `InMemoryProductRepository` reference. Notice the following lines in `ProductController`:

```
@Autowired
private ProductRepository productRepository;
```

Okay, but why is the `@Autowired` annotation here? If you observe the `ProductController` class carefully, you may wonder why we didn't instantiate any object for the `productRepository` reference. Nowhere could we see a single line saying something like `productRepository = new InMemoryProductRepository()`.

So how come executing the line `productRepository.getAllProducts()` works fine without any `NullPointerException` in the `list` method of the `ProductController` class?

```
model.addAttribute("products", productRepository.getAllProducts() );
```

Who is assigning the `InMemoryProductRepository` object to the `productRepository` reference? The answer is that the Spring framework is the one assigning the `InMemoryProductRepository` object to the `productRepository` reference.

Remember you learned that Spring would create and manage beans (objects) for every `@controller` class? Similarly, Spring would create and manage beans for every `@Repository` class as well. As soon as Spring sees the annotation `@Autowired` on top of the `ProductRepository` reference, it will assign an object of `InMemoryProductRepository` to that reference, since Spring already created and holds the `InMemoryProductRepository` object in its object container (web application context).

Remember we configured the component scan through the following annotation in the web application context configuration file:

```
@ComponentScan("com.packt.webstore")
```

And you learned earlier that if we configure our web application context with `@ComponentScan` annotation, it not only detects controllers (`@controller`), it also detects other stereotypes such as repositories (`@Repository`) and services (`@Service`) as well.

Since we added the `@Repository` annotation on top of the `InMemoryProductRepository` class, Spring knows that if any reference of the type `productRepository` has an `@Autowired` annotation on top of it, then it should assign the implementation object `InMemoryProductRepository` to that reference. This process of managing dependencies between classes is called **dependency injection** or **wiring** in the Spring world. So to mark any class as a repository object, we need to annotate that class with the `@Repository` (`org.springframework.stereotype.Repository`) annotation.

Okay, you understand how the Persistence layer works, but after the repository object returns a list of products, how do we show it in the web page? If you remember how we added our first product to the model, it is very similar to that instead of a single object. This time we are adding a list of objects to the model through the following line in the `list` method of `ProductController`:

```
model.addAttribute("products", productRepository.getAllProducts());
```

In the previous code, `productRepository.getAllProducts()` just returns a list of product domain objects (`List<Product>`) and we directly add that list to the model.

And in the corresponding view file (`products.jsp`), using the `<c:forEach>` tag, we loop through the list and show each product's information inside a styled `<div>` tag:

```
<c:forEach items="${products}" var="product">
<div class="col-sm-6 col-md-3" style="padding-bottom: 15px">
<div class="thumbnail">
    <div class="caption">
        <h3>${product.name}</h3>
        <p>${product.description}</p>
        <p>${product.unitPrice} USD</p>
    <p> Available ${product.unitsInStock} units in stock </p>
    </div>
</div>
</div>
</c:forEach>
```

Again, remember the `products` text in the `${products}` expression is nothing but the key that we used while adding the product list to the model from the `ProductController` class.

The for each loop is a special **JavaServer Pages Standard Tag Library (JSTL)** looping tag that will run through the list of products and assign each product to a variable called `product` (`var="product"`) on each iteration. From the `product` variable, we are fetching information such as the name, description, and price of the product and showing it within `<h3>` and `<p>` tags. That's how we are finally able to see the list of products in the products web page.



The JSTL is a collection of useful JSP tags that encapsulates the core functionality common to many JSP applications.

The Service layer

So far so good, we have created a Presentation layer that contains a controller, a dispatcher servlet, view resolvers, and more. And then we created the Domain layer, which contains a single domain class `Product`. Finally, we created the Persistence layer, which contains a repository interface and an implementation to access our `Product` domain objects from an in-memory database.

But we are still missing one more concept called the Service layer. Why do we need the Service layer? We have seen a Persistence layer dealing with all data access (CRUD) related logic, a Presentation layer dealing with all web requests and view-related activities, and a Domain layer containing classes to hold information that is retrieved from database records/the Persistence layer. But where can we put the business operations code?

The Service layer exposes business operations that could be composed of multiple CRUD operations. Those CRUD operations are usually performed by the repository objects. For example, you could have a business operation that would process a customer order, and in order to perform such a business operation, you would need to perform the following operations in order:

1. First, ensure that all the products in the requested order are available in your store.
2. Second, ensure there are sufficient quantities of those products in your store.
3. Finally, update the product inventory by reducing the available count for each product ordered.

Service objects are good candidates to put such business operation logic, where it requires multiple CRUD operations to be carried out by the repository layer for a single service call. The service operations could also represent the boundaries of SQL transactions meaning that all the elementary CRUD operations performed inside the business operations should be inside a transaction and either all of them should succeed, or they should roll back in the case of an error.

Time for action – creating a service object

Let's create a service object that will perform the simple business operation of updating stock. Our aim is dead simple: whenever we enter the URL

`http://localhost:8080/webstore/update/stock/`, our web store should go through the inventory of products and add 1,000 units to the existing stock if the number in stock is less than 500:

1. Open the `ProductRepository` interface from the `com.packt.webstore.domain.repository` package in the `src/main/java` source folder and add one more method declaration in it as follows:

```
void updateStock(String productId, long noOfUnits);
```

2. Open the `InMemoryProductRepository` implementation class and add an implementation for the previous declared method as follows:

```
@Override
public void updateStock(String productId, long noOfUnits) {
    String SQL = "UPDATE PRODUCTS SET UNITS_IN_STOCK =
:unitsInStock WHERE ID = :id";
    Map<String, Object> params = new HashMap<>();
    params.put("unitsInStock", noOfUnits);
    params.put("id", productId);
    jdbcTemplate.update(SQL, params);
}
```

3. Create an interface called `ProductService` under the `com.packt.webstore.service` package in the `src/main/java` source folder. And add a method declaration in it as follows:

```
void updateAllStock();
```

4. Create a class called `ProductServiceImpl` under the `com.packt.webstore.service.impl` package in the `src/main/java` source folder. And add the following code to it:

```
package com.packt.webstore.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation
.Autowired;
import org.springframework.stereotype.Service;

import com.packt.webstore.domain.Product;
import com.packt.webstore.domain.repository
.ProductRepository;
import com.packt.webstore.service.ProductService;

@Service
public class ProductServiceImpl implements ProductService{

    @Autowired
    private ProductRepository productRepository;
    @Override
    public void updateAllStock() {
        List<Product> allProducts =
        productRepository.getAllProducts();
        for(Product product : allProducts) {
            if (product.getUnitsInStock()<500)
                productRepository.updateStock
                (product.getProductid(),
                 product.getUnitsInStock()+1000);
        }
    }
}
```

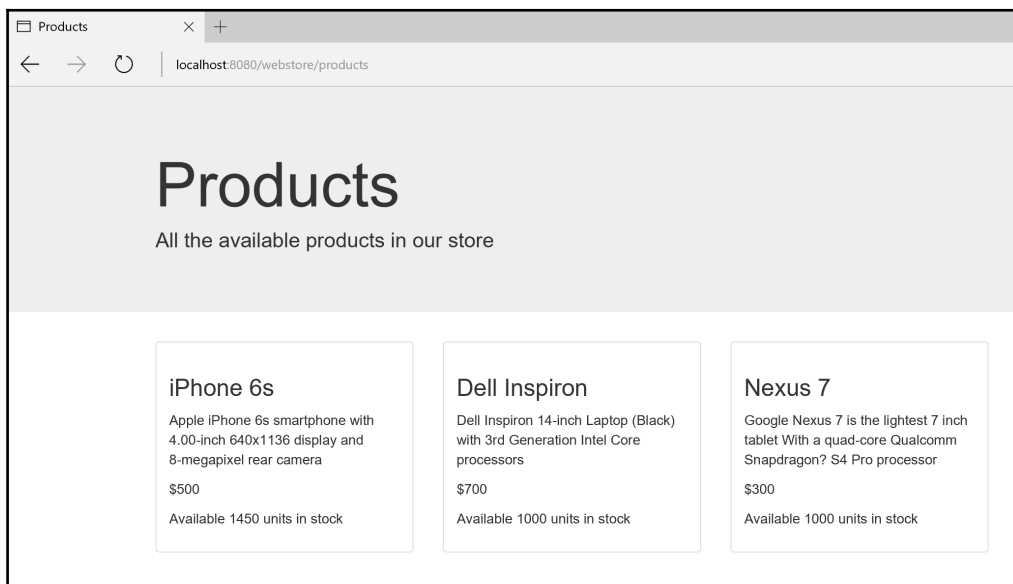
5. Open `ProductController` from the `com.packt.webstore.controller` package in the `src/main/java` source folder and add a private reference to `ProductService` with the `@Autowired` annotation as follows:

```
@Autowired
private ProductService productService;
```

6. Now add one more method definition as follows in `ProductController`:

```
@RequestMapping("/update/stock")
public String updateStock(Model model) {
    productService.updateAllStock();
    return "redirect:/products";
}
```

7. Run your application and enter the URL `http://localhost:8080/webstore/products`. You will be able to see a web page showing all the products. Notice the available units in stock for iPhone 6s will now show as **Available 450 units in stock**. All other products will show 1,000 as **Available 450 units in stock**.
8. Now enter the URL `http://localhost:8080/webstore/update/stock`, you will be able to see the same web page showing all the products. But this time, you can see that the available units in stock for iPhone 6s have been updated, and will show as **Available 1450 units in stock**.



Products page showing the product after stock has been updated via a service call

What just happened?

Okay, before going through the steps I just want to remind you of two facts regarding repository objects—that all the data access (CRUD) operations in a domain object should be carried out through repository objects only. Fact number two is that service objects rely on repository objects to carry out all data access related operations. That's why before creating the actual service interface/implementation, we created a repository interface/implementation method (`updateStock`) in steps 1 and 2.

The `updateStock` method from the `InMemoryProductRepository` class just updates a single product domain object's `unitsInStock` property for the given product. We need this method when we write logic for our service object method (`updateAllStock`) in the `OrderServiceImpl` class.

Now we come to steps 3 and 4 where we created the actual service definition and implementation. In step 3, we created an interface called `ProductService` to define all the expected responsibilities of an order service. As of now, we defined only one responsibility within that interface, which updates all the stock via the `updateAllStock` method. In step 4, we implemented the `updateAllStock` method within the `OrderServiceImpl` class, where we retrieved all the products and went through them one by one in a for loop to check whether the `unitsInStock` is less than 500. If so, we add 1,000 more units only to that product.

In the previous exercise, within the `ProductController` class, we connected to the repository through the `ProductRepository` interface reference to maximize loose coupling. Similarly, now we have connected the Service layer and repository layer through the `ProductRepository` interface reference as follows in the `ProductServiceImpl` class:

```
@Autowired
private ProductRepository productRepository;
```

As you already learned, Spring assigns the `InMemoryProductRepository` object to `productRepository` reference in the previously mentioned code because the `productRepository` reference has an `@Autowired` annotation and we know that Spring creates and manages all the `@Service` and `@Repository` objects. Remember that `OrderServiceImpl` has an `@Service` annotation on top of it.

To ensure transactional behavior, Spring provides an `@Transactional` (`org.springframework.transaction.annotation.Transactional`) annotation. We must annotate service methods with an `@Transactional` annotation to define transaction attributes, and we need to make some more configurations in our application context to ensure the transactional behavior takes effect.



Since our book is about Spring MVC and the Presentation layer, I omitted the `@Transactional` annotation from our Service layer objects. To find out more about transactional management in Spring, check out <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html>.

Okay, we have created the Service layer, and now it is ready to be consumed from the Presentation layer. It is time for us to connect our Service layer with the controller. In step 5, we created one more controller method called `updateStock` in `ProductController` to call the service object method:

```
@RequestMapping("/update/stock")
public String updateStock(Model model) {
    productService.updateAllStock();
    return "redirect:/products";
}
```

The `updateStock` method from `ProductController` class uses our `productService` reference to update all the stock.

You can also see that we mapped the `/update/stock` URL path to the `updateStock` method using the `@RequestMapping` annotation. So finally, when we are trying to hit the URL `http://localhost:8080/webstore/update/stock`, we are able to see the available units in stock being updated by 1,000 more units for the product `P1234`.

Have a go hero – accessing the product domain object via a service

In our `ProductController` class, we only have the `ProductRepository` reference to access the `Product` domain object within the `list` method. But accessing `ProductRepository` directly from the `ProductController` is not the best practice, as it is always good to access the Persistence layer repository via a service object.

Why don't you create a Service layer method to mediate between `ProductController` and `ProductRepository`? Here are some of things you can try out:

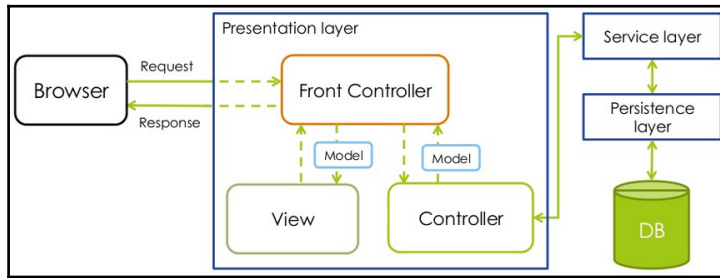
- Create a method declaration as `List <Products> getAllProducts()` within the `ProductService` interface
- Create an implementation method for the previous method declaration in `ProductServiceImpl`
- Use the `ProductRepository` reference within the `getAllProducts` method of `ProductServiceImpl` to get all the products from `ProductRepository`
- Remove the `ProductRepository` reference in the `ProductController` class and accordingly change the `list` method in `ProductController`

After finishing this, you will be able to see the same product listings under the URL `http://localhost:8080/webshop/products/` without any problems.

An overview of the web application architecture

So far you have seen how to organize our code into layers so that we can avoid tight coupling between codes and improve reusability and separation of concerns. We just created one domain class, one repository class, and one service class to demonstrate a purpose, but a typical real-world MVC application may contain many domain, repository, and service classes. Each layer is usually connected through interfaces, and the controller always accesses domain objects from the repository via a service interface only.

So every typical enterprise-level Spring MVC application will logically have four layers, namely Presentation, Domain, Persistence, and Services. The Domain layer is sometimes called the model layer. The following block diagram will help you conceptualize this idea:



Layers of a Spring MVC application

So you have learned how to create a Service layer object and a repository layer object, but what you saw in the Service layer and repository layer was just a glimpse. Spring has extensive support for database and transaction handling, which is a vast topic and deserves its own book. So in the upcoming chapters, we will concentrate more on the Presentation layer, which contains more Spring MVC related concepts, rather than the database and transaction-related concepts.

Have a go hero – listing all our customers

It's great that we have listed all our products in our web application under the URL `http://localhost:8080/webstore/products`, but in order to become a successful web store, maintaining only the product information is not enough—we need to maintain information about the customers as well, so that we can attract them by giving them special discounts based on their purchase history.

So why don't you maintain customer information in your application too? Here are some improvements you can make to your application to maintain customer information as well as product information:

- Add one more `Customer` domain class in the same package where `Product` exists:
 - Add fields such as `customerId`, `name`, `address`, and `noOfOrdersMade` to the `Customer` class
- Create a Persistence layer to return all the customers
 - Create an interface called `CustomerRepository` with a method declaration such as `List <Customers> getAllCustomers()`
 - Create an `InMemoryCustomerRepository` implementation for `CustomerRepository` and instantiate some dummy customer in the constructor of `InMemoryCustomerRepository` likes we did in `InMemoryProductRepository`
- Create a Service layer to get all the customers from the repository
 - Create an interface called `CustomerService` with a method declaration such as `List <Customers> getAllCustomers()`
 - Create an implementation `CustomerServiceImpl` for `CustomerService`
- Create one more controller called `CustomerController`
 - Add a request mapping method to map the URL `http://localhost:8080/webstore/customers`
- Create a view file called `customers.jsp`

After finishing this exercise, you will be able to see all your customers under the URL `http://localhost:8080/webstore/customers`. It is very similar to the way we listed all our products under the URL `http://localhost:8080/webstore/products`.

Summary

At the start of this chapter, you learned about the duties of the Dispatcher servlet and how it maps requests using the `@RequestMapping` annotation. Next you saw what the web application context is and how to configure a web application context for our web application. After that, you had a brief introduction to view resolvers and how `InternalResourceViewResolver` resolves the view file for a given logical view name.

You also learned the concept of MVC and saw the overall request flow of a Spring MVC application. Then you learned about the web application architecture. In the web application architecture section, you saw how to create and organize code under the various layers of a Spring MVC application such as the Domain layer, the Persistence layer, and the Service layer. During its course, we showed you how to retrieve product domain objects from the repository and present them on a web page using the controller. You also learned where a service object fits in. Finally, you got an overview of the web application architecture.

We hope you now have a good overall understanding about Spring MVC and the various components involved in developing a Spring MVC application. In the next chapter, you will learn more about controllers and related concepts in depth. So see you in the next chapter!

3

Control Your Store with Controllers

In Chapter 2, Spring MVC Architecture – Architecting Your Web Store you saw the overall architecture of a Spring MVC application. We didn't go into any of the concepts in depth; our aim was for you to understand the overall flow.

In Spring MVC, the concept of Controllers has an important role, so we are going to look at Controllers in-depth in this chapter. This chapter will cover concepts such as:

- Defining a Controller
- URI template patterns
- Matrix variables
- Request parameters

The role of a Controller in Spring MVC

In Spring MVC, the Controller's methods are the final destination point that a web request can reach. After being invoked, the Controller's method starts to process the web request by interacting with the Service layer to complete whatever work needs to be done. Usually, the Service layer executes business operations on domain objects and calls the Persistence layer to update the domain objects. After the processing is completed by the Service layer object, the Controller is responsible for updating and building up the `model` object and chooses a View for the user to see next as a response.

Remember that Spring MVC always keeps the Controllers unaware of any View technology used. That's why the Controller returns only the logical View name, and later `DispatcherServlet` consults with `ViewResolver` to find out the exact View to render. According to the Controller, the Model is a collection of arbitrary Java objects and the View is identified by a logical View name. Rendering the correct View for the given logical View name is the responsibility of `ViewResolver`.

In all our previous exercises, Controllers are used to return the logical View name and the Model is updated via the `model` parameter available in the controller method. There is another, seldom used way of updating the `model` parameter and returning the View name from the controller parameter with the help of the `ModelAndView` (`org.springframework.web.servlet.ModelAndView`) object. Look at the following code snippets as an example:

```
@RequestMapping("/all")
public ModelAndView allProducts() {
    ModelAndView modelAndView = new ModelAndView();

    modelAndView.addObject("products", productService.getAllProducts());

    modelAndView.setViewName("products");

    return modelAndView;
}
```

This code snippet just shows you how to encapsulate the Model and View using the `modelAndView` object.

Defining a Controller

Controllers are the Presentation layer components that are responsible for responding to the user's actions. These actions could be entering a particular URL in the browser, clicking on a link, submitting a form on a web page, or something similar. Any regular Java classes can be transformed into a Controller by simply annotating them with the `@Controller` (`org.springframework.stereotype.Controller`) annotation.

And as you have already learned, the `@Controller` annotation supports Spring's component scanning mechanism in auto-detecting/registering the bean definition in the web application's context. To enable this auto-registering capability, we must add the `@ComponentScan` (`org.springframework.context.annotation.ComponentScan`) annotation in the web application context configuration file. You saw how to do this in Chapter 2, *Spring MVC Architecture – Architecting Your Web Store* under the section

Understanding the web application context configuration.

A Controller class is made up of request-mapped methods, also in short called handler methods, and handler methods are annotated with the `@RequestMapping` (`org.springframework.web.bind.annotation.RequestMapping`) annotation. The `@RequestMapping` annotation is used to map the request path of a URL to a particular handler method. In Chapter 2, *Spring MVC Architecture – Architecting Your Web Store* you got a short introduction to the `@RequestMapping` annotation and learned how to apply the `@RequestMapping` annotation on the handler method level, but in Spring MVC you can even specify the `@RequestMapping` annotation on the Controller class level. In that case, Spring MVC will consider the Controller class level `@RequestMapping` annotation's value before mapping the remaining URL request path to the handler methods. This feature is called relative request mapping.



The terms **request mapped method**, **mapped method**, **handler method**, and **controller method** all have the same meaning; these terms are used to specify a controller method with an `@RequestMapping` annotation. These terms are used interchangeably in this book.

Time for action – adding class-level request mapping

Let's add an `@RequestMapping` annotation on the class level of our `ProductController` to demonstrate the relative request mapping feature on the handler methods. But before that, we just want to ensure that you have already replaced the `ProductRepository` reference with the `ProductService` reference in the `ProductController` class as part of the previous chapter's *Have a go hero – accessing the product domain object via a service* section. Because contacting the Persistence layer directly from Presentation layer is not best practice, all access to the Persistence layer should go through the Service layer. Those who completed the exercise can directly start from step 5; others should continue from step 1.

1. Open the `ProductService` interface from the `com.packt.webstore.service` package in the `src/main/java` source folder, and add the following method declarations to it:

```
List<Product> getAllProducts();
```

2. Open the corresponding `ProductServiceImpl` implementation class from the `com.packt.webstore.service.impl` package in the `src/main/java` source folder, and add the following method implementation to it:

```
@Override
```

```
public List<Product> getAllProducts() {  
    return productRepository.getAllProducts();  
}
```

3. Open `ProductController` and remove the existing `ProductRepository` reference. Basically delete the following two lines from `ProductController`:

```
@Autowired  
private ProductRepository productRepository;
```

4. Now alter the body of the `list` method as follows in the `ProductController` class. Note this time we used the `productService` reference to get all the products:

```
@RequestMapping("/products")  
public String list(Model model) {  
    model.addAttribute("products",  
        productService.getAllProducts());  
  
    return "products";  
}
```

5. In the `ProductController` class, add the following annotation on top of the class:

```
@RequestMapping("market")
```

6. Again run our application and enter our regular URL to list the products (<http://localhost:8080/webstore/products/>). You will see the **HTTP Status 404** error page in the browser.
7. But now you can access the same products page under <http://localhost:8080/webstore/market/products/>.

What just happened?

What we demonstrated here is a simple concept called relative request mapping. In step 5, we just added an extra `@RequestMapping` annotation at the class level with a value attribute defined as `market`. Basically, your class signature will look as follows after your change:

```
@Controller  
@RequestMapping("market")  
public class ProductController {
```

In all our previous examples, we annotated `@RequestMapping` annotations only on the Controller's method level, but Spring MVC also allows us to specify request mapping on the Controller's class level. In that case, Spring MVC maps a specific URL request path on the method level that is relative to the class level of the `@RequestMapping` URL value. So if we have defined any class level request mapping, Spring MVC will consider that class level request path before mapping the remaining request path to the handler methods.

That's why when we accessed our regular products page in step 6, we got the **HTTP Status 404** error. Because we added an extra request mapping on the Controller class level, our list method is now being mapped to the URL

`http://localhost:8080/webstore/market/products`.

Now have we changed our Controller's request mapping in `ProductController`, we also need to change the return value of the `updateStock` method as follows to avoid side effects because of our change:

```
return "redirect:/market/products";
```

We will consider more about the `redirect:` prefix in the next chapter. For now, just remember this change is needed to avoid side-effects since we changed the Controller request mapping.

Default request mapping method

Every Controller class can have one default request mapping method. What is the default request mapping method? If we simply don't specify any request path value in the `@RequestMapping` annotation of a Controller's method, that method is designated as the default request mapping method for that class. So whenever a request URL just ends up with the Controller's class level request path value without any further relative path down the line, then Spring MVC will invoke this method as a response to that request.



If you specify more than one default mapping method inside a Controller, you will get `IllegalStateException` with the message **Ambiguous mapping found**. So a Controller can have only one default request mapping method at most.

Why not change the `welcome` method to the default request mapping method for our `HomeController` class?

1. In the `HomeController` class, add the following annotation on top of the class:

```
@RequestMapping("/")
```

2. And from the `welcome` method's `@RequestMapping` annotation, remove the `value` attribute completely, so now the `welcome` method will have a plain `@RequestMapping` annotation without any attributes, as follows:

```
@RequestMapping
public String welcome(Model model) {
```

3. Now you can access the same `welcome` page under `http://localhost:8080/webstore/`.

Pop quiz – class level request mapping

If you imagine a web application called *Library* with the following request mapping on a Controller class level and in the method level, which is the appropriate request URL to map the request to the `productDetails` method?

```
@RequestMapping("/books")
public class BookController {
    ...

    @RequestMapping(value = "/list")
    public String books(Model model) {
        ...
    }
}
```

1. `http://localhost:8080/library/books/list`
2. `http://localhost:8080/library/list`
3. `http://localhost:8080/library/list/books`
4. `http://localhost:8080/library/`

Similarly, suppose we have another handler method called `bookDetails` under `BookController` as follows, what URL will map to that method?

```
@RequestMapping
public String details(Model model) {
    ...

    1. http://localhost:8080/library/books/details
    2. http://localhost:8080/library/books
    3. http://localhost:8080/library/details
    4. http://localhost:8080/library/
```

Handler mapping

You have learned that `DispatcherServlet` is the thing that dispatches the request to the handler methods based on the request mapping, but in order to interpret the mappings defined in a request mapping, `DispatcherServlet` needs a `HandlerMapping` (`org.springframework.web.servlet.HandlerMapping`) implementation.

`DispatcherServlet` consults with one or more `HandlerMapping` implementations to know which handler can handle the request. So `HandlerMapping` determines which Controller to call.

`HandlerMapping` interface provides the abstraction for mapping requests to handlers. The `HandlerMapping` implementations are able to inspect the request and come up with an appropriate Controller. Spring MVC provides many `HandlerMapping` implementations, and the one we are using to detect and interpret mappings from the `@RequestMapping` annotation is the `RequestMappingHandlerMapping`

(`org.springframework.web.servlet.mvc.method.annotation.`

`RequestMappingHandlerMapping`) class. To start using

`RequestMappingHandlerMapping`, we have to add the `@EnableWebMvc` annotation in our web application context configuration file, so that Spring MVC can create and register a bean for `RequestMappingHandlerMapping` in our web application context. If you remember, we already configured the `@EnableWebMvc` annotation in Chapter 2, *Spring MVC Architecture – Architecting Your Web Store* under the *Understanding web application context configuration* section.

Using URI template patterns

In the previous chapters, you saw how to map a particular URL to a Controller's method; for example, if we entered the URL

`http://localhost:8080/webstore/market/products`, we would map that request to the `list` method of `ProductController` and list all the product information in the web page.

What if we want to list only a subset of products based on category, for instance, if we want to display only the products that fall under the category of laptops? If the URL entered is `http://localhost:8080/webstore/market/products/Laptop`, and similarly if the URL is `http://localhost:8080/webstore/market/products/Tablet`, we will like to show only tablets on the web page.

One way to do this is to have a separate request mapping method in the Controller for every unique category. But it won't scale if we have hundreds of categories; in that case we have to write hundreds of request mapping methods in the Controller. So how do we do that in an elegant way?

The Spring MVC URI template patterns feature is the answer. If you look at the following URLs carefully, the only part of the URL that changes is the category type (`Laptop` and `Tablet`). Other than that, everything is the same:

- `http://localhost:8080/webstore/products/Laptop`
- `http://localhost:8080/webstore/products/Tablet`

So we can define a common URI template for these URLs, which might look like `http://localhost:8080/webstore/products/{category}`. Spring MVC can leverage this fact and make the template portion (`{category}`) of the URL a variable, which is called a path variable in the Spring MVC world.

Time for action – showing products based on category

Let's add a category-wise View to our products page using the path variable.

1. Open the `ProductRepository` interface and add one more method declaration to `getProductsByCategory`:

```
List<Product> getProductsByCategory(String category);
```

2. Open the `InMemoryProductRepository` implementation class and add an implementation for the previously declared method as follows:

```
@Override
public List<Product> getProductsByCategory(String category) {
    String SQL = "SELECT * FROM PRODUCTS WHERE CATEGORY =
:category";
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("category", category);

    return jdbcTemplate.query(SQL, params, new
    ProductMapper());
}
```

3. Similarly open the `ProductService` interface and add one more method declaration to `getProductsByCategory`:

```
List<Product> getProductsByCategory(String category);
```

4. Open the `ProductServiceImpl` service implementation class and add an implementation as follows:

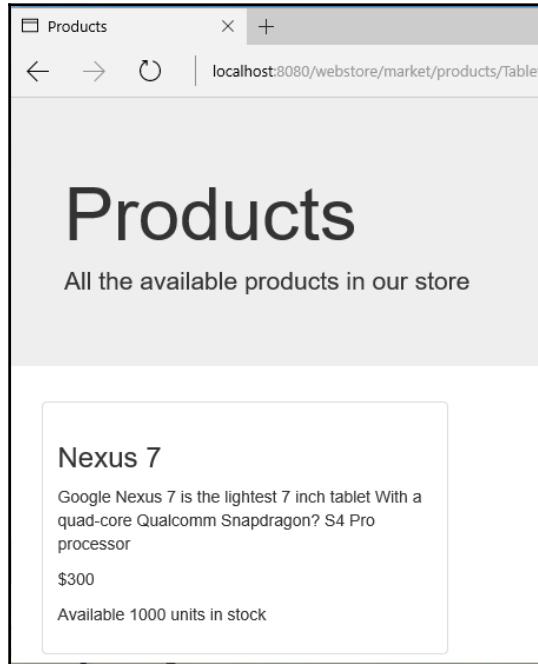
```
public List<Product> getProductsByCategory(String category) {
    return productRepository.getProductsByCategory(category);
}
```

5. Now open our `ProductController` class and add one more request mapping method as follows:

```
@RequestMapping("/products/{category}")
public String getProductsByCategory(Model model,
    @PathVariable("category") String productCategory) {
    model.addAttribute("products",
    productService.getProductsByCategory(productCategory));
    return "products";
}
```

6. Now run our application and enter the following URL:

`http://localhost:8080/webstore/market/products/Tablet`. You should see the following screen:



Screen showing products by category with the help of path variables

What just happened?

Step 5 is the most important in the whole sequence, because all the steps prior to step 5 are a prerequisite for it. What we are doing in step 5 is nothing but one normal way of adding a list of product objects to the `model`.

```
model.addAttribute("products",  
    productService.getProductsByCategory(productCategory));
```

One thing you need to notice here is the `getProductsByCategory` method from `productService`; we need this method to get the list of products for the given category. And `productService` as such cannot give the list of products for the given category; it will ask the repository.

That's why in step 4 we used the `productRepository` reference to get the list of products by category in the `ProductServiceImpl` class. Notice the following line from `ProductServiceImpl`:

```
return productRepository.getProductsByCategory(category);
```

Another important thing you need to notice in the step 5 code snippet is the `@RequestMapping` annotation's request path value:

```
@RequestMapping("/products/{category}")
```

By enclosing a portion of a request path within curly braces, we are indicating to Spring MVC that it is a URI template variable. According to the Spring MVC documentation, a URI template is a URI-like string containing one or more variable names. When you substitute values for these variables, the template becomes a URI.

For example, the URI template

`http://localhost:8080/webstore/market/products/{category}` contains the category variable. Assigning the value `laptop` to the variable yields `http://localhost:8080/webstore/market/products/Laptop`. In Spring MVC, we can use the `@PathVariable` (`org.springframework.web.bind.annotation.PathVariable`) annotation to read a URI template variable.

Since we have the `@RequestMapping("/market")` annotation on the `ProductController` level, the actual request path for the `getProductsByCategory` method will be `/market/products/{category}`. So at runtime, if we provide a web request URL such as `http://localhost:8080/webstore/market/products/Laptop`, then the category path variable will have the value `laptop`. Similarly for the web request `http://localhost:8080/webstore/market/products/Tablet`, the category path variable will have the value `tablet`.

Now how do we retrieve the value stored in the URI template path variable `category`? As we already mentioned, the `@PathVariable` annotation will help us to read that variable. All we need to do is simply annotate the `getProductsByCategory` method's parameter with the `@PathVariable` annotation as follows:

```
public String getProductsByCategory(@PathVariable("category") String  
productCategory, Model model) {
```

Spring MVC will read whatever value is present in the `category` URI template variable and assign it to the `productCategory` method parameter. So we have the category value in a variable; we just pass it to `productService` to get the list of products in that category. Once we get that list of products, we simply add it to the Model and return the same View name that we used to list all the products.

The value attribute in the `@PathVariable` annotation should be the same as the variable name in the path expression of the `@RequestMapping` annotation. For example, if the path expression is `"/products/{identity}"`, then to retrieve the path variable `identity` you have to form the `@PathVariable` annotation as `@PathVariable("identity")`.

If the `@PathVariable` annotation is specified without any value attribute, it will try to retrieve a path variable with the name of the variable it has been annotated with.



For example, if you specify simply `@PathVariable String productId`, then Spring will assume that it should look for a URI template variable `{productId}` in the URL. A request mapping method can have any number of `@PathVariable` annotations.

Finally in step 6, when we enter the URL

`http://localhost:8080/webstore/market/products/Tablet`, we see information about Google's Nexus 7, which is a tablet. Similarly, if you enter the URL

`http://localhost:8080/webstore/products/Laptop`, you will be able to see Dell's Inspiron laptop's information.

Pop quiz – request path variable

If we have a web application called `webstore` with the following request mapping on the Controller class level and in the method level, which is the appropriate request URL?

```
@RequestMapping("/items")
public class ProductController {
    ...
    @RequestMapping(value = "/type/{type}", method = RequestMethod.GET)
    public String productDetails(@PathVariable("type") String productType,
        Model model) {
```

1. `http://localhost:8080/webstore/items/electronics`
2. `http://localhost:8080/webstore/items/type/electronics`

3. `http://localhost:8080/webstore/items/productType/electronics`
4. `http://localhost:8080/webstore/type/electronics`

For the following request mapping annotation, which are the correct methods' signatures to retrieve the path variables?

```
@RequestMapping(value="/manufacturer/{  
manufacturerId}/product/{productId}")
```

1. `public String productByManufacturer(@PathVariable String manufacturerId, @PathVariable String productId, Model model)`
2. `public String productByManufacturer (@PathVariable String manufacturer, @PathVariable String product, Model model)`
3. `public String productByManufacturer
(@PathVariable("manufacturer") String manufacturerId,
@PathVariable("product") String productId, Model model)`
4. `public String productByManufacturer
(@PathVariable("manufacturerId") String manufacturer,
@PathVariable("productId") String product, Model model)`

Using matrix variables

In the previous section, you saw the URI template facility to bind variables in the URL request path. But there is one more way to bind variables in the request URL in a name-value pair style, referred to as matrix variables within Spring MVC. Look at the following URL:

```
http://localhost:8080/webstore/market/products/filter/price;low=500;high=1000
```

In this URL, the actual request path is just up to

`http://localhost:8080/webstore/market/products/filter/price`, and after that we have something like `low=500;high=1000`. Here, `low` and `high` are just matrix variables. But what makes Matrix variables so special is the ability to assign multiple values for a single variable; that is, we can assign a list of values to a URI variable. Look at the following URL:

```
http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop
```


In this URL, we have two variables, namely `brand` and `category`. Both have multiple values: `brands` (`Google`, `Dell`) and `categories` (`Tablet`, `Laptop`). How can we read these variables from the URL during request mapping? Here comes the special binding annotation `@MatrixVariable` (`org.springframework.web.bind.annotation.MatrixVariable`). One cool thing about the `@MatrixVariable` annotation is that it allows us to collect the matrix variables in the map of a collection (`Map<String, List<String>>`), which will be more helpful when we are dealing with complex web requests.

Time for action – showing products based on filters

Consider a situation where we want to filter the product list based on `brands` and `categories`. For example, you want to list all the products that fall under the category `Laptop` and `Tablets` and from the manufacturer `Google` and `Dell`. With the help of `Matrix` variables, we can form a URL something like the following to bind the `brands` and `categories` variables' values into the URL:

```
http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop
```

Let's see how to map this URL to a handler method with the help of the `@MatrixVariable` annotation:

1. Open the `ProductRepository` interface and add one more method declaration to `getProductsByFilter`:

```
List<Product> getProductsByFilter (Map<String,List<String>> filterParams);
```

2. Open the `InMemoryProductRepository` implementation class and add the following method implementation for `getProductsByFilter`:

```
@Override
public List<Product> getProductsByFilter (Map<String,
List<String>> filterParams) {
    String SQL = "SELECT * FROM PRODUCTS WHERE CATEGORY IN (
        :categories ) AND MANUFACTURER IN ( :brands)";

    return jdbcTemplate.query(SQL, filterParams, new
        ProductMapper());
}
```

3. Open the `ProductService` interface and add one more method declaration to `getProductsByFilter`:

```
List<Product> getProductsByFilter(Map<String, List<String>>
filterParams);
```

4. Open the `ProductServiceImpl` service implementation class and add the following method implementation for `getProductsByFilter`:

```
public List<Product> getProductsByFilter(Map<String,
List<String>> filterParams) {
    return productRepository.getProductsByFilter(filterParams);
}
```

5. Open `ProductController` and add one more request mapping method as follows:

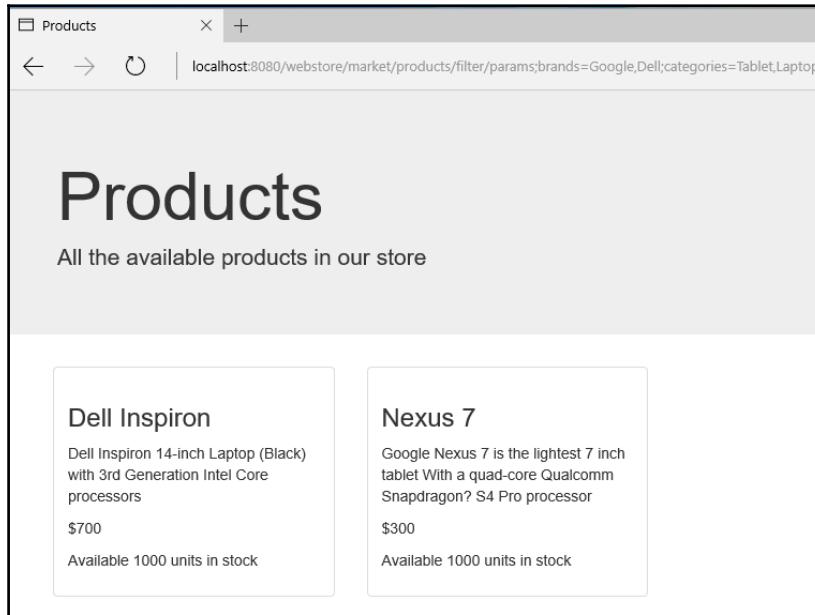
```
@RequestMapping("/products/filter/{params}")
public String
getProductsByFilter(@MatrixVariable(pathVar="params")
Map<String,List<String>> filterParams, Model model) {
    model.addAttribute("products",
    productService.getProductsByFilter(filterParams));
    return "products";
}
```

6. Open our web application context configuration file (`WebApplicationContextConfig.java`) and enable matrix variable support by overriding the `configurePathMatch` method as follows:

```
@Override
public void configurePathMatch(PathMatchConfigurer
configurer) {
    UrlPathHelper urlPathHelper = new UrlPathHelper();
    urlPathHelper.setRemoveSemicolonContent(false);

    configurer.setUrlPathHelper(urlPathHelper);
}
```

7. Now run our application and enter the following URL:
`http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop`. You will see the products list as shown in the following screen:



Using matrix variables to show the product list filtered by criteria

What just happened?

Our aim was to retrieve the matrix variable values from the URL and do something useful; in our case the URL we were trying to map is `http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop`. Here we want to extract the matrix variables `brands` and `categories`. The `brands` and `categories` variables have values: (Google, Dell) and (Tablet, Laptop) respectively. In the previously specified URL, the request path is just up to `http://localhost:8080/webstore/market/products/filter/params` only. That's why in step 5 we annotated our `getProductsByFilter` request mapping method as follows:

```
@RequestMapping("/products/filter/{params}")
```

But you may be wondering why we have a URI template (`{params}`) in the `@RequestMapping` annotation as a mapping to a path variable. This is because, if our request URL contains a matrix variable, then we have to form the `@RequestMapping` annotation with a URI template to identify the matrix variable segments. That's why we defined `params` as a URI template in the request mapping `@RequestMapping("/products/filter/{params}")` annotation.



A URL can have multiple matrix variables, and each matrix variable must be separated with a `“;”` (semicolon). To assign multiple values to a single variable, each value must be `“,”` (comma) separated or we can repeat the variable name. See the following URL, which is a variable repeated version of the same URL that we used in our example:

`http://localhost:8080/webstore/market/products/filter/params;brands=Google;brands=Dell;categories=Tablet;categories=Laptop`

Note that we repeated the variable `brands` and `categories` twice in the URL.

Okay, we mapped the web request to the `getProductsByFilter` method, but how do we retrieve the value from the matrix variables? The answer is the `@MatrixVariable` annotation.

`@MatrixVariable` is very similar to the `@PathVariable` annotation; if you look at the `getProductsByFilter` method signature in step 5, we annotated the method's parameter `filterParams` with the `@MatrixVariable` annotation as follows:

```
public String getProductsByFilter(@MatrixVariable(pathVar="params")
    Map<String,List<String>> filterParams, Model model)
```

So Spring MVC will read all the matrix variables found in the URL after the `{params}` URI template and put them into the method parameter `filterParams` map. The `filterParams` map will have each matrix variable name as the key and the corresponding list will contain multiple values assigned for the matrix variable. The `pathVar` attribute from `@MatrixVariable` is used to identify the matrix variable segment in the URL; that's why it has the value `params`, which is nothing but the URI template value we used in our request mapping URL.

A URL can have multiple matrix variable segments. See the following URL:

`http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop/specification;dimention=10,20,15;color=red,green,blue`

It contains two matrix variable segments each identified by the prefix `params` and `specification` respectively. So, in order to capture each matrix variable segment into maps, we have to form the controller method signature as follows:

```
@RequestMapping("/products/filter/{params}/{specification}")
public String filter(@MatrixVariable(pathVar="params")
    Map<String,List<String>> criteriaFilter, @MatrixVariable(pathVar="
specification") Map<String,List<String>> specFilter, Model model)
```

Okay, we got the value of the matrix variables' values into the method parameter `filterParams`, but what we have done with that `filterParams` map? We simply passed it as parameters to the service method to retrieve the products based on the criteria:

```
productService.getProductsByFilter(filterParams)
```

Again, the service passes that map to the repository to get the list of products based on the criteria. Once we get the list, as usual, we simply add that list to the `Model`, and return the same logical View name that was used to list the products.

To enable the use of matrix variables in Spring MVC, we must set the `RemoveSemicolonContent` property of `UrlPathHelper` to `false`; we did that in step 6. Finally, we are able to see products based on the specified criteria in step 7 on our product listing page.

Understanding request parameters

Matrix variables and path variables are a great way to bind variables in the URL request path. However, there is one more way to bind variables in the HTTP request, not only as a part of the URL but also in the body of the HTTP web request; these are the so-called HTTP parameters. You might have heard about GET or POST parameters; GET parameters have been used for years as a standard way to bind variables in URLs, and POST is used to bind variables in the body of an HTTP request. You will learn about POST parameters in the next chapter during form submission.

Okay, now let's see how to read GET request parameters in the Spring MVC style. To demonstrate the usage of a request parameter, let's add a product details page to our application.

Time for action – adding a product detail page

So far in our product listing page, we have only shown product information such as the product's name, description, price, and available units in stock. But we haven't shown information such as the manufacturer, category, product ID, and more. Let's add a product detail page to show them:

1. Open the `ProductRepository` interface from the `com.packt.webstore.domain.repository` package in the `src/main/java` source folder and add one more method declaration on it as follows:

```
Product getProductById(String productID);
```

2. Open the `InMemoryProductRepository` implementation class and add an implementation for the previously declared method as follows:

```
@Override
public Product getProductById(String productID) {
    String SQL = "SELECT * FROM PRODUCTS WHERE ID = :id";
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("id", productID);

    return jdbcTemplate.queryForObject(SQL, params, new
ProductMapper());
}
```

3. Open the `ProductService` interface and add one more method declaration to it as follows:

```
Product getProductById(String productID);
```

4. Open the `ProductServiceImpl` service implementation class and add the following method implementation for `getProductById`:

```
@Override
public Product getProductById(String productID) {
    return productRepository.getProductById(productID);
}
```

5. Open our `ProductController` class and add one more request mapping method as follows:

```
@RequestMapping("/product")
public String getProductById(@RequestParam("id") String
productId, Model model) {
    model.addAttribute("product",
```

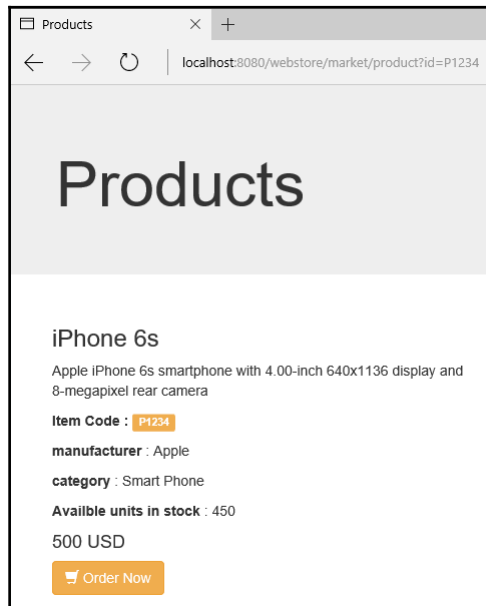
```
productService.getProductById(productId) );  
    return "product";  
}
```

6. And finally add one more JSP View file called `product.jsp` under the `src/main/webapp/WEB-INF/views/` directory, add the following code snippets into it, and save it:

```
<%@ taglib prefix="c"  
uri="http://java.sun.com/jsp/jstl/core"%>  
  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html;  
charset=ISO-8859-1">  
<link rel="stylesheet"  
  
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/  
bootstrap.min.css">  
<title>Products</title>  
</head>  
<body>  
    <section>  
        <div class="jumbotron">  
            <div class="container">  
                <h1>Products</h1>  
            </div>  
        </div>  
    </section>  
    <section class="container">  
        <div class="row">  
            <div class="col-md-5">  
                <h3>${product.name}</h3>  
                <p>${product.description}</p>  
                <p>  
                    <strong>Item Code : </strong><span  
                        class="label label warning">${product.productId}  
                </span>  
                </p>  
                <p>  
                    <strong>manufacturer</strong> :  
                    ${product.manufacturer}  
                </p>  
                <p>  
                    <strong>category</strong> :  
                    ${product.category}  
                </p>  
                <p>
```

```
        <strong>Availble units in stock </strong> :  
        ${product.unitsInStock}  
    </p>  
    <h4>${product.unitPrice} USD</h4>  
    <p>  
        <a href="#" class="btn btn-warning btn-large">  
        <span class="glyphicon-shopping-cart glyphicon">  
</span> Order Now  
        </a>  
    </p>  
    </div>  
</div>  
</section>  
</body>  
</html>
```

7. Now run our application and enter the following URL:
`http://localhost:8080/webstore/market/product?id=P1234`. You will
be able to see the product detail page as shown in the following screenshot:



Using request parameters to show the product detail page

What just happened?

In steps 1 and 2, we just created a repository method declaration/implementation to get products for the given product ID (`getProductById`). Similarly in steps 3 and 4, we created a corresponding Service layer method declaration and implementation to access the `getProductById` method. What we did in step 5 is very similar to what we did in the `getProductsByCategory` method of `ProductController`. We just added a product object to the `model` that is returned by the service object:

```
model.addAttribute("product", productService.getProductById(productId));
```

But here, the important question is, who is providing the value for the `productId` parameter? The answer is simple, as you guessed; since we annotated the parameter `productId` with `@RequestParam("id")` annotation (`org.springframework.web.bind.annotation.RequestParam`), Spring MVC will try to read a GET request parameter with the name `id` from the URL and will assign that to the `getProductById` method's parameter `productId`.

The `@RequestParam` annotation also follows the same convention as other binding annotations; that is, if the name of the GET request parameter and the name of the variable it is annotating are the same, then there is no need to specify the value attribute in the `@RequestParam` annotation.

And finally in step 6, we added one more View file called `product.jsp`, because we want a detailed view of the product where we can show all the information about the product. Nothing fancy in this `product.jsp` file; as usual we are getting the value from the `model` and showing it within HTML tags using the usual JSTL expression notation `${}`:

```
<h3>${product.name}</h3>
  <p>${product.description}</p>
  ... ..
```

Okay, you saw how to retrieve a GET request parameter from an URL, but how do you pass more than one GET request parameter in the URL? The answer is simple: the standard HTTP protocol defines a way for it; we simply need to delimit each parameter value pair with an `&` symbol; for example, if you want to pass `category` and `price` as GET request parameters in a URL, you have to form the URL as follows:

```
http://localhost:8080/webstore/product?category=laptop&price=700
```

Similarly, to map this URL in a request mapping method, your request mapping method should have at least two parameters with the `@RequestParam` annotation annotated:

```
public String getProducts(@RequestParam String category, @RequestParam
String price) {
```

Pop quiz – the request parameter

For the following request mapping method signature, which is the appropriate request URL?

```
@RequestMapping(value = "/products", method = RequestMethod.GET)
public String productDetails(@RequestParam String rate, Model model)
```

1. `http://localhost:8080/webstore/products/rate=400`
2. `http://localhost:8080/webstore/products?rate=400`
3. `http://localhost:8080/webstore/products?rate/400`
4. `http://localhost:8080/webstore/products/rate=400`

Time for action – implementing a master detail View

A master detail View is nothing but a display of very important information in a master page; once we select an item in the master View, a detailed page of the selected item will be shown in the detailed View page. Let's build a master detail View for our product listing page, so that, when we click on any product, we can see the detailed View of that product.

We have already implemented the product listing page

(`http://localhost:8080/webstore/market/products`) and product detail page (`http://localhost:8080/webstore/market/product?id=P1234`), so the only thing we need to do is connect those two Views to make it a master detail View. Let's see how to do that:

1. Open `products.jsp`. You can find `products.jsp` under `src/main/webapp/WEB-INF/views/`. Add the following spring tag lib reference on top of the file:

```
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
```

2. Add the following lines after the Available units in stock paragraph tag in `products.jsp`:

```
<p>
<a href=" <spring:url value="/market/product?
id=${product.productId}" /> " class="btn btn-primary">
<span class="glyphicon-info-sign glyphicon"/></span> Details
</a>
</p>
```

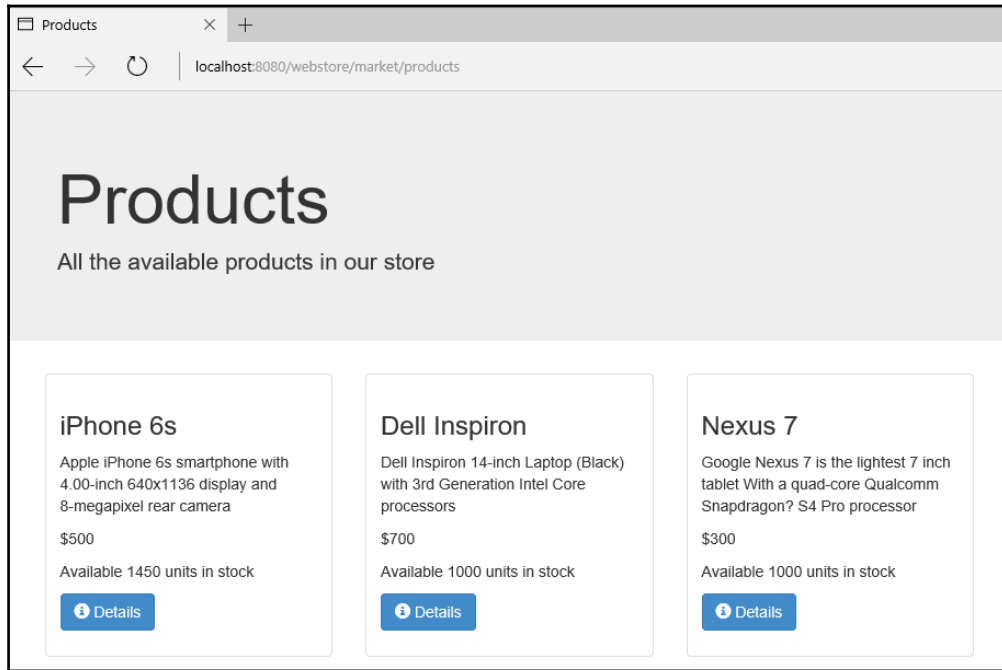
3. Now open `product.jsp`. You can find `product.jsp` under `src/main/webapp/WEB-INF/views/`. Add the following spring tag lib reference on top of the file:

```
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
```

4. And, add the following lines just before the Order Now link in `product.jsp`:

```
<a href="<spring:url value="/market/products" />" class="btn
btn-default">
<span class="glyphicon-hand-left glyphicon"></span> back
</a>
```

5. Now run our application and enter the following URL `http://localhost:8080/webstore/market/products`. You will be able to see the product list page, and every product will have a **Details** button as shown in the following screenshot:



Master View of product listings

- Now click on the any product's **Details** button and you will be able to see the detailed view with the back button link to the product listing page.

What just happened?

What we did is simple; in step 2 we just added a hyperlink using the following tag in `products.jsp`:

```
<a href=" <spring:url value="/market/product?id=${product.productId}" /> "
htmlEscape="true" />" class="btn btn-primary">
  <span class="glyphicon-info-sign glyphicon"/></span> Details
</a>
```

Notice the `href` attribute of a `<a>` tag, which has a `<spring:url>` tag as its value:

```
<spring:url value="/market/product?id=${product.productId}" />
```

This `<spring:url>` tag is used to construct a valid Spring URL. We need this `<spring:url>` to be used in step 2; that's why we added reference to the Spring tag library in step 1. Observe the value attribute of the `<spring:url>` tag, and you can see that, for the `id` URL parameter, we assigned the expression `${product.productId}`. So during the rendering of this link, Spring MVC will assign the corresponding product ID in that expression.

For example, while rendering the link for the first product, Spring MVC will assign the value `P1234` for the product ID. So the final URL value with `<spring:url>` in it will become `/market/product?id=P1234`, which is nothing but the request mapping path for the product's details page. So when you click this link, you will land on the details page for that particular product.

Similarly, we need a link back to the product listing page from product detail page; that's why we added another link in the `product.jsp` tag as follows in step 4:

```
<a href="<spring:url value="/market/products" />" class="btn btn-default">
  <span class="glyphicon-hand-left glyphicon"></span> back
</a>
```

Note the `` tag is just to style the button with an icon, so you do not need to pay attention to it too much; the only interesting thing for us is the `href` attribute of the `<a>` tag, which has the `<spring:url>` tag with the value attribute `/market/products` in it.

Have a go hero – adding multiple filters to list products

It is good that you learned various techniques to bind parameters with URLs such as using path variables, matrix variables, and GET parameters. We saw how to get the products of a particular category using path variables, how to get products within a particular brand and category using the matrix variable, and finally how to get a particular product by the product ID using a request parameter.

Now imagine you want to apply multiple criteria to view a desired product; for example, what if you want to view a product that falls under the `Tablet` category, and within the price range of \$200 to \$400, and from the manufacturer `Google`?

To retrieve a product that can satisfy all this criteria, we can form a URL something like this:

```
http://localhost:8080/webstore/products/Tablet/price;low=200;high=400?b
rand="Google"
```

Why don't you write a corresponding controller method to serve this request URL? Here are some hints to accomplish this requirement:

- Create one more request mapping method called `filterProducts` in the `productController` class to map the following URL:

```
http://localhost:8080/webstore/products/Tablet/price;low=200  
;high=400?brand="Google"
```

Remember this URL contains matrix variables `low` and `high` to represent the price range, a GET parameter called `brand` to identify the manufacturer, and finally a URI template path variable called `Tablet` to represent the category.

- You can use the same View file `products.jsp` to list the filtered products.

Good luck!

Summary

In this chapter, you learned about the role of a Controller in Spring MVC first. Next you learned how to define a Controller and saw the usage of the `@Controller` annotation. After that, you learned the concept of relative request mapping, where you saw how to define request mapping on the Controller level and understood how Spring relatively maps a web request to the Controller's request mapping method. You also learned how the Dispatcher servlet uses handler mapping to find out the exact handler methods. You saw various parameter binding techniques such as URI template patterns, matrix variables, and HTTP GET request parameters to bind parameter with URL. Finally, you saw how to implement a master detail View.

In the next chapter, we are going explore various Spring tags that are available in the Spring tag library. You will also learn more about form processing and how to bind form data with the HTTP POST parameter. Get ready for the next chapter...

4

Working with Spring Tag Libraries

You learned that one of the biggest advantages of using Spring MVC is its ability to separate View technologies from the rest of the MVC framework. Spring MVC supports various View technologies such as JSP/JSTL, Thymeleaf, Tiles, FreeMarker, Velocity, and more. In previous chapters, you saw some basic examples of how to use `InternalResourceViewResolver` to implement JSP/JSTL views. Spring MVC provides first-class support for JSP/JSTL views with the help of Spring tag libraries. In this chapter, you are going to learn more about the various tags that are available as part of Spring tag libraries.

After finishing this chapter, you will have a good idea about the following topics:

- **JavaServer Pages Standard Tag Library (JSTL)**
- Serving and processing web forms
- Form-binding and whitelisting
- Spring tag libraries

The JavaServer Pages Standard Tag Library

JavaServer Pages (JSP) is a technology that lets you embed Java code inside HTML pages. This code can be inserted by means of `<% %>` blocks or by means of JSTL tags. To insert Java code into JSP, JSTL tags are generally preferred, since tags adapt better to their own tag representation of HTML, making JSP pages look more readable.



JSP even lets you define your own tags; you must write the code that actually implements the logic of your own tags in Java.

JSTL is just a standard tag library provided by Oracle. We can add a reference to the JSTL tag library in our JSP pages as follows:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

Similarly, Spring MVC also provides its own tag library to develop Spring JSP views easily and effectively. These tags provide a lot of useful common functionality such as form binding, evaluating errors, and outputting messages, and more when we work with Spring MVC.

In order to use these, Spring MVC has provided tags in our JSP pages. We must add a reference to that tag library in our JSP pages as follows:

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

These `taglib` directives declare that our JSP page uses a set of custom tags related to Spring and identify the location of the library. They also provide a means to identify custom tags in our JSP page. In the `taglib` directive, the `uri` attribute value resolves to a location that the servlet container understands and the `prefix` attribute announces which bits of markup are custom actions.

Serving and processing forms

In the previous chapters, you learned how to retrieve data from an in-memory database using the Controller, but you didn't learn how to store the data in an in-memory database from the View. In Spring MVC, the process of putting an HTML form element's values into model data is called form binding.

In all the previous chapter's examples, you saw that the data transfer took place from the Model to the View via the Controller. The following line is a typical example of how we put data into the Model from the Controller:

```
model.addAttribute(greeting, "Welcome")
```


Similarly, the next line shows how we retrieve that data in the View using a JSTL expression:

```
<p> ${greeting} </p>
```

But what if we want to put data into the Model from the View? How do we retrieve that data in the Controller? For example, consider a scenario where an admin of our store wants to add new product information to our store by filling out and submitting an HTML form. How can we collect the values filled out in the HTML form elements and process them in the Controller? This is where Spring tag library tags help us to bind the HTML tag element's values to a form backing bean in the Model. Later, the Controller can retrieve the form backing bean from the Model using the `@ModelAttribute` (`org.springframework.web.bind.annotation.ModelAttribute`) annotation.



The form backing bean (sometimes called the form bean) is used to store form data. We can even use our domain objects as form beans; this works well when there's a close match between the fields in the form and the properties in our domain object. Another approach is creating separate classes for form beans, which is sometimes called **Data Transfer Objects (DTO)**.

Time for action – serving and processing forms

The Spring tag library provides some special `<form>` and `<input>` tags, which are more or less similar to HTML form and input tags, but have some special attributes to bind form elements' data with the form backed bean. Let's create a Spring web form in our application to add new products to our product list:

1. Open our `ProductRepository` interface and add one more method declaration to it as follows:

```
void addProduct(Product product);
```

2. Add an implementation for this method in the `InMemoryProductRepository` class as follows:

```
@Override
public void addProduct(Product product) {
    String SQL = "INSERT INTO PRODUCTS (ID, "
        + "NAME, "
        + "DESCRIPTION, "
        + "UNIT_PRICE, "
        + "MANUFACTURER, "
```

```
        + "CATEGORY, "
        + "CONDITION, "
        + "UNITS_IN_STOCK, "
        + "UNITS_IN_ORDER, "
        + "DISCONTINUED) "
        + "VALUES (:id, :name, :desc, :price,
            :manufacturer, :category, :condition, :inStock,
            :inOrder, :discontinued)";
Map<String, Object> params = new HashMap<>();
params.put("id", product.getProductId());
params.put("name", product.getName());
params.put("desc", product.getDescription());
params.put("price", product.getUnitPrice());
params.put("manufacturer", product.getManufacturer());
params.put("category", product.getCategory());
params.put("condition", product.getCondition());
params.put("inStock", product.getUnitsInStock());
params.put("inOrder", product.getUnitsInOrder());
params.put("discontinued", product.isDiscontinued());

jdbcTemplate.update(SQL, params);
}
```

3. Open our `ProductService` interface and add one more method declaration to it as follows:

```
void addProduct(Product product);
```

4. And add an implementation for this method in the `ProductServiceImpl` class as follows:

```
@Override
public void addProduct(Product product) {
    productRepository.addProduct(product);
}
```

5. Open our `ProductController` class and add two more request mapping methods as follows:

```
@RequestMapping(value = "/products/add", method =
RequestMethod.GET)
public String getAddNewProductForm(Model model) {
    Product newProduct = new Product();
    model.addAttribute("newProduct", newProduct);
    return "addProduct";
}
@RequestMapping(value = "/products/add", method =
```

```
RequestMethod.POST)
public String
processAddNewProductForm(@ModelAttribute("newProduct")
Product newProduct) {
    productService.addProduct(newProduct);
    return "redirect:/market/products";
}
```

6. Finally, add one more JSP View file called `addProduct.jsp` under the `src/main/webapp/WEB-INF/views/` directory and add the following tag reference declaration as the very first line in it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form" %>
```

7. Now add the following code snippet under the tag declaration line and save `addProduct.jsp`. Note that I skipped some `<form:input>` binding tags for some of the fields of the product domain object, but I strongly encourage you to add binding tags for the skipped fields while you are trying out this exercise:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/
bootstrap.min.css">
<title>Products</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Products</h1>
                <p>Add products</p>
            </div>
        </div>
    </section>
    <section class="container">
        <form:form method="POST" modelAttribute="newProduct"
class="form-horizontal">
            <fieldset>
                <legend>Add new product</legend>

                <div class="form-group">
```

```
        <label class="control-label col-lg-2 col-lg-2"
            for="productId">Product Id</label>
        <div class="col-lg-10">
            <form:input id="productId" path="productId"
                type="text" class="form-input-large"/>
        </div>
    </div>

    <!-- Similarly bind <form:input> tag for
        name,unitPrice,manufacturer,category,unitsInStock
    and unitsInOrder fields-->

    <div class="form-group">
        <label class="control-label col-lg-2"
            for="description">Description</label>
        <div class="col-lg-10">
            <form:textarea id="description"
                path="description" rows = "2"/>
        </div>
    </div>

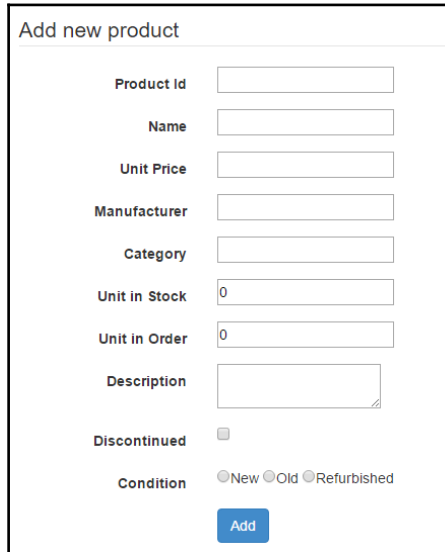
    <div class="form-group">
        <label class="control-label col-lg-2"
            for="discontinued">Discontinued</label>
        <div class="col-lg-10">
            <form:checkbox id="discontinued"
                path="discontinued"/>
        </div>
    </div>

    <div class="form-group">
        <label class="control-label col-lg-2"
            for="condition">Condition</label>
        <div class="col-lg-10">
            <form:radiobutton path="condition"
                value="New" />New
            <form:radiobutton path="condition"
                value="Old" />Old
            <form:radiobutton path="condition"
                value="Refurbished" />Refurbished
        </div>
    </div>

    <div class="form-group">
        <div class="col-lg-offset-2 col-lg-10">
            <input type="submit" id="btnAdd" class="btn
                btn-primary" value ="Add"/>
        </div>
    </div>
</fieldset>
```

```
        </form:form>
    </section>
</body>
</html>
```

8. Now run our application and enter the URL:
<http://localhost:8080/webstore/market/products/add>. You will be able to see a web page showing a web form to add product information as shown in the following screenshot:



The screenshot shows a web form titled "Add new product". It contains the following fields and controls:

- Product Id:
- Name:
- Unit Price:
- Manufacturer:
- Category:
- Unit in Stock:
- Unit in Order:
- Description:
- Discontinued: ☐
- Condition: ☐ New ☐ Old ☐ Refurbished
- At the bottom is a blue button labeled "Add".

Add a products web form

9. Now enter all the information related to the new product that you want to add and click on the **Add** button. You will see the new product added in the product listing page under the URL
<http://localhost:8080/webstore/market/products>.

What just happened?

In the whole sequence, steps 5 and 6 are very important steps and need to be observed carefully. Everything mentioned prior to step 5 was very familiar to you I guess. Anyhow, I will give you a brief note on what we did in steps 1 to 4.

In step 1, we just created an `addProduct` method declaration in our `ProductRepository` interface to add new products. And in step 2, we just implemented the `addProduct` method in our `InMemoryProductRepository` class. Steps 3 and 4 are just a Service layer extension for `ProductRepository`. In step 3, we declared a similar `addProduct` method in our `ProductService` and implemented it in step 4 to add products to the repository via the `productRepository` reference.

Okay, coming back to the important step; all we did in step 5 was add two request mapping methods, namely `getAddNewProductForm` and `processAddNewProductForm`:

```
@RequestMapping(value = "/products/add", method = RequestMethod.GET)
public String getAddNewProductForm(Model model) {
    Product newProduct = new Product();
    model.addAttribute("newProduct", newProduct);
    return "addProduct";
}

@RequestMapping(value = "/products/add", method = RequestMethod.POST)
public String processAddNewProductForm(@ModelAttribute("newProduct")
Product productToBeAdded) {
    productService.addProduct(productToBeAdded);
    return "redirect:/market/products";
}
```

If you observe those methods carefully, you will notice a peculiar thing: both the methods have the same URL mapping value in their `@RequestMapping` annotations (`value = "/products/add"`). So if we enter the URL

`http://localhost:8080/webstore/market/products/add` in the browser, which method will Spring MVC map that request to?

The answer lies in the second attribute of the `@RequestMapping` annotation (`method = RequestMethod.GET` and `method = RequestMethod.POST`). Yes if you look again, even though both methods have the same URL mapping, they differ in the request method.

So what is happening behind the scenes is that, when we enter the URL `http://localhost:8080/webstore/market/products/add` in the browser, it is considered as a GET request, so Spring MVC will map that request to the `getAddNewProductForm` method. Within that method, we simply attach a new empty `Product` domain object with the model, under the attribute name `newProduct`. So in the `addproduct.jsp` View, we can access that `newProduct` Model object:

```
Product newProduct = new Product();
model.addAttribute("newProduct", newProduct);
```

Before jumping into the `processAddNewProductForm` method, let's review the `addproduct.jsp` View file in some detail, so that you understand the form processing flow without confusion. In `addproduct.jsp`, we just added a `<form:form>` tag from Spring's tag library:

```
<form:form modelAttribute="newProduct" class="form-horizontal">
```

Since this special `<form:form>` tag is coming from a Spring tag library, we need to add a reference to that tag library in our JSP file; that's why we added the following line at the top of the `addProducts.jsp` file in step 6:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

In the Spring `<form:form>` tag, one of the important attributes is `modelAttribute`. In our case, we assigned the value `newProduct` as the value of `modelAttribute` in the `<form:form>` tag. If you remember correctly, you can see that this value of the `modelAttribute`, and the attribute name we used to store the `newProduct` object in the Model from our `getAddNewProductForm` method, are the same. So the `newProduct` object that we attached to the model from the Controller method (`getAddNewProductForm`) is now bound to the form. This object is called the form backing bean in Spring MVC.

Okay, now you should look at every `<form:input>` tag inside the `<form:form>` tag. You can observe a common attribute in every tag. That attribute is `path`:

```
<form:input id="productId" path="productId" type="text" class="form:input-large"/>
```

The `path` attribute just indicates the field name that is relative to the form backing bean. So the value that is entered in this input box at runtime will be bound to the corresponding field of the form bean.

Okay, now it's time to come back and review our `processAddNewProductForm` method. When will this method be invoked? This method will be invoked once we press the **submit** button on our form. Yes, since every form submission is considered a POST request, this time the browser will send a POST request to the same URL `http://localhost:8080/webstore/products/add`.

So this time the `processAddNewProductForm` method will get invoked since it is a POST request. Inside the `processAddNewProductForm` method, we are simply calling the `addProduct` service method to add the new product to the repository:

```
productService.addProduct (productToBeAdded) ;
```

But the interesting question here is, how come the `productToBeAdded` object is populated with the data that we entered in the form? The answer lies in the `@ModelAttribute` (`org.springframework.web.bind.annotation.ModelAttribute`) annotation. Notice the method signature of the `processAddNewProductForm` method:

```
public String processAddNewProductForm(@ModelAttribute("newProduct")
Product productToBeAdded)
```

Here if you look at the value attribute of the `@ModelAttribute` annotation, you can observe a pattern. Yes, the `@ModelAttribute` annotation's value and the value of the `modelAttribute` from the `<form:form>` tag are the same. So Spring MVC knows that it should assign the form bounded `newProduct` object to the `processAddNewProductForm` method's `productToBeAdded` parameter.

The `@ModelAttribute` annotation is not only used to retrieve an object from the Model, but if we want we can even use the `@ModelAttribute` annotation to add objects to the Model. For instance, we can even rewrite our `getAddNewProductForm` method to something like the following, using the `@ModelAttribute` annotation:

```
@RequestMapping(value = "/products/add", method = RequestMethod.GET)
public String getAddNewProductForm(@ModelAttribute("newProduct") Product
newProduct) {
    return "addProduct";
}
```

You can see that we haven't created a new empty `Product` domain object and attached it to the model. All we did was add a parameter of the type `Product` and annotated it with the `@ModelAttribute` annotation, so Spring MVC will know that it should create an object of `Product` and attach it to the model under the name `newProduct`.

One more thing that needs to be observed in the `processAddNewProductForm` method is the logical View name it is returning: `redirect:/market/products`. So what we are trying to tell Spring MVC by returning the string `redirect:/market/products`? To get the answer, observe the logical View name string carefully; if we split this string with the ":" (colon) symbol, we will get two parts. The first part is the prefix `redirect` and the second part is something that looks like a request path: `/market/products`. So, instead of returning a View name, we are simply instructing Spring to issue a redirect request to the request path `/market/products`, which is the request path for the `list` method of our `ProductController`. So, after submitting the form, we list the products using the `list` method of `ProductController`.



As a matter of fact, when we return any request path with the `redirect :` prefix from a request mapping method, Spring will use a special View object called `RedirectView` (`org.springframework.web.servlet.view.RedirectView`) to issue the redirect command behind the scenes. We will see more about `RedirectView` in the next chapter.

Instead of landing on a web page after the successful submission of a web form, we are spawning a new request to the request path `/market/products` with the help of `RedirectView`. This pattern is called `redirect-after-post`; it is a commonly used pattern with web-based forms. We are using this pattern to avoid double submission of the same form.



Sometimes after submitting the form, if we press the browser's refresh button or back button, there are chances to resubmit the same form. This behavior is called `double submission`.

Have a go hero – customer registration form

It is great that we created a web form to add new products to our web application under the URL `http://localhost:8080/webstore/market/products/add`. Why don't you create a customer registration form in our application to register a new customer in our application? Try to create a customer registration form under the URL `http://localhost:8080/webstore/customers/add`.

Customizing data binding

In the last section, you saw how to bind data submitted by an HTML form to a form backing bean. In order to do the binding, Spring MVC internally uses a special binding object called `WebDataBinder` (`org.springframework.web.bind.WebDataBinder`).

`WebDataBinder` extracts the data out of the `HttpServletRequest` object and converts it to a proper data format, loads it into a form backing bean, and validates it. To customize the behavior of data binding, we can initialize and configure the `WebDataBinder` object in our Controller. The `@InitBinder` (`org.springframework.web.bind.annotation.InitBinder`) annotation helps us to do that. The `@InitBinder` annotation designates a method to initialize `WebDataBinder`.

Let's look at a practical use for customizing `WebDataBinder`. Since we are using the actual domain object itself as a form backing bean, during the form submission there is a chance of security vulnerabilities. Because Spring automatically binds HTTP parameters to form bean properties, an attacker could bind a suitably-named HTTP parameter with form properties that weren't intended for binding. To address this problem, we can explicitly tell Spring which fields are allowed for form binding. Technically speaking, the process of explicitly telling which fields are allowed for binding is called whitelisting binding in Spring MVC; we can do whitelisting binding using `WebDataBinder`.

Time for action – whitelisting form fields for binding

In the previous exercise, while adding a new product we bound every field of the `Product` domain in the form, but it is meaningless to specify `unitsInOrder` and `discontinued` values during the addition of a new product because nobody can make an order before adding the product to the store; similarly `discontinued` products need not be added in our product list. So we should not allow these fields to be bound with the form bean while adding a new product to our store. However we should only allow all the other fields of the `Product` domain object to be bound. Let's see how to do this with the following steps:

1. Open our `ProductController` class and add a method as follows:

```
@InitBinder
public void initialiseBinder(WebDataBinder binder) {
    binder.setAllowedFields("productId",
        "name",
        "unitPrice",
        "description",
        "manufacturer",
        "category",
        "unitsInStock",
        "condition");
}
```

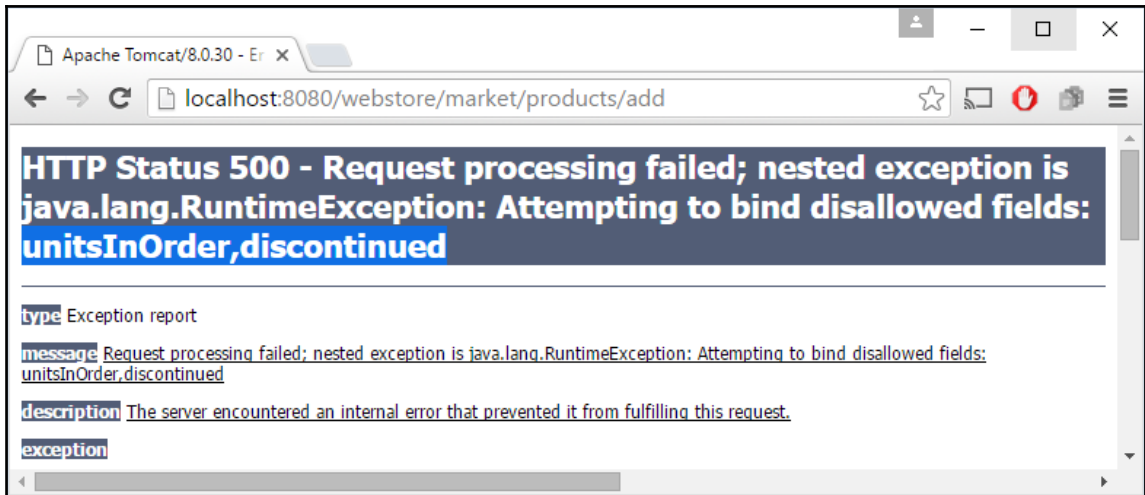
2. Add an extra parameter of the type `BindingResult` (`org.springframework.validation.BindingResult`) to the `processAddNewProductForm` method as follows:

```
public String
processAddNewProductForm(@ModelAttribute("newProduct")
    Product productToBeAdded, BindingResult result)
```

3. In the same `processAddNewProductForm` method, add the following condition just before calling the `productService.addProduct(newProduct)` line:

```
String[] suppressedFields = result.getSuppressedFields();
if (suppressedFields.length > 0) {
    throw new RuntimeException("Attempting to bind
        disallowed fields: " +
        StringUtils.arrayToCommaDelimitedString(suppressedFields));
}
```

4. Now run our application and enter the URL `http://localhost:8080/webstore/market/products/add`. You will be able to see a web page showing a web form to add new product information. Fill out all the fields, particularly `Units in order` and `discontinued`.
5. Now press the **Add** button and you will see a **HTTP Status 500** error on the web page as shown in the following image:



The add product page showing an error for disallowed fields

6. Now open `addProduct.jsp` from `src/main/webapp/WEB-INF/views/` in your project and remove the input tags that are related to the `Units in order` and `discontinued` fields. Basically, you need to remove the following block of code:

```
<div class="form-group">
    <label class="control-label col-lg-2"
        for="unitsInOrder">Units In
```

```
Order</label>
<div class="col-lg-10">
    <form:input id="unitsInOrder" path="unitsInOrder"
        type="text" class="form-input-large"/>
</div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2"
        for="discontinued">Discontinued</label>
    <div class="col-lg-10">
        <form:checkbox id="discontinued" path="discontinued"/>
    </div>
</div>
```

7. Now run our application again and enter the URL `http://localhost:8080/webstore/market/products/add`. You will be able to see a web page showing a web form to add a new product, but this time without the **Units in order** and **Discontinued** fields.
8. Now enter all information related to the new product and click on the **Add** button. You will see the new product added in the product listing page under the URL `http://localhost:8080/webstore/market/products`.

What just happened?

Our intention was to put some restrictions on binding HTTP parameters with the form baking bean. As we already discussed, the automatic binding feature of Spring could lead to a potential security vulnerability if we used a domain object itself as form bean. So we have to explicitly tell Spring MVC which are fields are allowed. That's what we are doing in step 1.

The `@InitBinder` annotation designates a Controller method as a hook method to do some custom configuration regarding data binding on `WebDataBinder`. And `WebDataBinder` is the thing that is doing the data binding at runtime, so we need to specify which fields are allowed to bind to `WebDataBinder`. If you observe our `initialiseBinder` method from `ProductController`, it has a parameter called `binder`, which is of the type `WebDataBinder`. We are simply calling the `setAllowedFields` method on the `binder` object and passing the field names that are allowed for binding. Spring MVC will call this method to initialize `WebDataBinder` before doing the binding since it has the `@InitBinder` annotation.



`WebDataBinder` also has a method called `setDisallowedFields` to strictly specify which fields are disallowed for binding. If you use this method, Spring MVC allows any HTTP request parameters to be bound except those field names specified in the `setDisallowedFields` method. This is called blacklisting binding.

Okay, we configured which fields are allowed for binding, but we need to verify whether any fields other than those allowed are bound with the form baking bean. That's what we are doing in steps 2 and 3.

We changed `processAddNewProductForm` by adding one extra parameter called `result`, which is of the type `BindingResult`. Spring MVC will fill this object with the result of the binding. If any attempt is made to bind any fields other than the allowed fields, the `BindingResult` object will have a `getSuppressedFields` count greater than zero. That's why we were checking the suppressed field count and throwing a `RuntimeException` exception:

```
if (suppressedFields.length > 0) {  
    throw new RuntimeException("Attempting to bind disallowed fields: " +  
        StringUtils.arrayToCommaDelimitedString(suppressedFields));  
}
```



Here the static class `StringUtils` comes from `org.springframework.util.StringUtils`.

We want to ensure that our binding configuration is working; that's why we run our application without changing the View file `addProduct.jsp` in step 4. And as expected, we got the **HTTP Status 500** error saying **Attempting to bind disallowed fields** when we submitted the **Add products** form with the `unitsInOrder` and `discontinued` fields filled out. Now we know our binder configuration is working, we could change our View file so as not to bind the disallowed fields. That's what we were doing in step 6: just removing the input field elements that are related to the disallowed fields from the `addProduct.jsp` file.

After that, our added new products page works just fine, as expected. If any outside attackers try to tamper with the POST request and attach a HTTP parameter with the same field name as the form baking bean, they will get a `RuntimeException`.

The whitelisting is just an example of how can we customize the binding with the help of `WebDataBinder`. But by using `WebDataBinder`, we can perform many more types of binding customization as well. For example, `WebDataBinder` internally uses many `PropertyEditor` (`java.beans.PropertyEditor`) implementations to convert the HTTP request parameters to the target field of the form backing bean. We can even register custom `PropertyEditor` objects with `WebDataBinder` to convert more complex data types. For instance, look at the following code snippet that shows how to register the custom `PropertyEditor` to convert a `Date` class:

```
@InitBinder
public void initialiseBinder (WebDataBinder binder) {
    DateFormat dateFormat = new SimpleDateFormat("MMM d, YYYY");
    CustomDateEditor orderDateEditor = new CustomDateEditor(dateFormat,
true);
    binder.registerCustomEditor(Date.class, orderDateEditor);
}
```

There are many advanced configurations we can make with `WebDataBinder` in terms of data binding, but for a beginner level we don't need that level of detail.

Pop quiz – data binding

Consider the following data binding customization and identify the possible matching field bindings:

```
@InitBinder
public void initialiseBinder(WebDataBinder binder) {
    binder.setAllowedFields("unit*");
}
```

1. NoOfUnit
2. unitPrice
3. priceUnit
4. united

Externalizing text messages

So far, in all our View files we hardcoded text values for all the labels. As an example, take our `addProduct.jsp` file—for the `productId` input tag, we have a label tag with the hardcoded text value as `Product id`:

```
<label class="control-label col-lg-2 col-1g-2" for="productId">Product  
Id</label>
```

Externalizing these texts from a View file into a properties file will help us to have a single centralized control for all label messages. Moreover, it will help us to make our web pages ready for internationalization. We will talk more about internationalization in Chapter 6, *Internalize Your Store with Interceptor*, but, in order to perform internalization, we need to externalize the label messages first. So now you are going to see how to externalize locale-sensitive text messages from a web page to a property file.

Time for action – externalizing messages

Let's externalize the labels texts in our `addProduct.jsp`:

1. Open our `addProduct.jsp` file and add the following tag lib reference at the top:

```
<%@ taglib prefix="spring"  
uri="http://www.springframework.org/tags" %>
```

2. Change the `productId`'s `<label>` tag value `Product Id` to `<spring:message code="addProduct.form.productId.label"/>`. After changing your `productId`'s `<label>` tag value, it should look as follows:

```
<label class="control-label col-lg-2 col-1g-2"  
for="productId"> <spring:message  
code="addProduct.form.productId.label"/> </label>
```

3. Create a file called `messages.properties` under `/src/main/resources` in your project and add the following line to it:

```
addProduct.form.productId.label = New Product ID
```

4. Now open our web application context configuration file `WebApplicationContextConfig.java` and add the following bean definition to it:

```
@Bean  
public MessageSource messageSource() {  
    ResourceBundleMessageSource resource = new  
        ResourceBundleMessageSource();  
    resource.setBasename("messages");  
    return resource;  
}
```

5. Now run our application again and enter the URL `http://localhost:8080/webstore/market/products/add`. You will be able to see the added product page with the product ID label showing as **New Product ID**.

What just happened?

Spring MVC has a special tag called `<spring:message>` to externalize texts from JSP files. In order to use this tag, we need to add a reference to a Spring tag library; that's what we did in step 1. We just added a reference to the Spring tag library in our `addProduct.jsp` file:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

In step 2, we just used that tag to externalize the label text of the product ID input tag:

```
<label class="control-label col-lg-2 col-lg-2" for="productId">  
  <spring:message code="addProduct.form.productId.label"/> </label>
```

Here, an important thing you need to remember is the `code` attribute of the `<spring:message>` tag; we have assigned the value `addProduct.form.productId.label` as the code for this `<spring:message>` tag. This `code` attribute is a kind of key; at runtime Spring will try to read the corresponding value for the given key (`code`) from a message source property file.

We said that Spring will read the message's value from a message source property file, so we need to create that property file. That's what we did in step 3. We just created a property file with the name `messages.properties` under the resource directory. Inside that file, we just assigned the label text value to the message tag `code`:

```
addProduct.form.productId.label = New Product ID
```

Remember, for demonstration purposes I just externalized a single label, but a typical web application will have externalized messages for almost all tags; in that case `messages.properties` file will have many code-value pair entries.

Okay, we created a message source property file and added the `<spring:message>` tag in our JSP file, but to connect these two we need to create one more Spring bean in our web application context for the `org.springframework.context.support.ResourceBundleMessageSource` class with the name `messageSource`—we did that in step 4:


```
@Bean
public MessageSource messageSource() {
    ResourceBundleMessageSource resource = new
    ResourceBundleMessageSource();
    resource.setBasename("messages");
    return resource;
}
```

One important property you need to notice here is the `basename` property; we assigned the value `messages` for that property. If you remember, this is the name of the property file that we created in step 3.

That is all we did to enable the externalizing of messages in a JSP file. Now if we run the application and open up the **Add products** page, you can see that the product ID label will have the same text as we assigned to the `addProdcut.form.productId.label` code in the `messages.properties` file.

Have a go hero – externalizing all the labels from all the pages

I just showed you how to externalize the message for a single label; you can now do that for every single label available in all the pages.

Summary

At the start of this chapter, you saw how to serve and process forms, and you learned how to bind form data with a form backing bean. You also learned how to read a bean in the Controller. After that, we went a little deeper into form bean binding and configured the binder in our Controller to whitelist some of the POST parameters from being bound to the form bean. Finally, you saw how to use one more Spring special tag `<spring:message>` to externalize the messages in a JSP file.

In the next chapter, you will find out more about Views and view resolvers.

5

Working with View Resolver

In the last chapter, you saw how to use some of the Spring tags that could only be used in JSP and JSTL Views, but Spring has excellent support for other View technologies as well. Spring MVC maintains a high level of decoupling between the View and the Controller; the Controller knows nothing about the View except the View name. It is the responsibility of the view resolver to map the correct View for the given View name.

In this chapter, we will have a deeper look into Views and view resolvers. After finishing this chapter, you will have a clear idea about:

- Views and resolving Views
- Static Views
- A multipart view resolver
- Content negotiation
- Handler Exception Resolver

Resolving Views

As I already mentioned, Spring MVC does not make any assumptions about specific View technologies. According to Spring MVC, a View is identifiable as an implementation of the `org.springframework.web.servlet.View` interface:

```
public interface View {

    String getContentType();

    void render(Map<String, ?> model, HttpServletRequest request,
        HttpServletResponse response) throws Exception;
}
```

The render method from the Spring MVC View interface defines, as the main responsibility for a view object, that it should render proper content as a response (`javax.servlet.http.HttpServletResponse`) based on the given Model and request (`javax.servlet.http.HttpServletRequest`).

Because of the simplicity of the Spring MVC View interface, if we want we can write our own View implementation. But Spring MVC provides many convenient View implementations that are ready to use by simply configuring them in our web application context configuration file.

One such View is `InternalResourceView` (`org.springframework.web.servlet.view.InternalResourceView`) for rendering a response as a JSP page. Similarly, there are other View implementations such as `RedirectView`, `TilesView`, `FreeMarkerView`, `VelocityView`, and more available for specific View technologies. Spring MVC does not encourage you to couple the view object with the Controller as it will lead the controller method to tightly couple with one specific View technology. But if you want to do so, you can do something like the following code snippet:

```
@RequestMapping("/home")
public ModelAndView greeting(Map<String, Object> model) {
    model.put("greeting", "Welcome to Web Store!");
    model.put("tagline", "The one and only amazing web store");
    View view = new InternalResourceView("/WEB-INF/views/welcome.jsp");
    return new ModelAndView(view, model);
}
```

In this code handler method, we didn't return any logical View name; rather we instantiated `InternalResourceView` out of `welcome.jsp` directly and composed it into the `ModelAndView` (`org.springframework.web.servlet.ModelAndView`) object. This example is not encouraged since it tightly coupled the greeting handler method with `InternalResourceView`. Instead, what we can do is return a logical View name and configure an appropriate view resolver of our choice in our web app context to create a view object.

Spring comes with quite a few view resolvers to resolve various type of Views. You already saw how to configure `InternalResourceViewResolver` as our view resolver to resolve JSP Views in Chapter 2, *Spring MVC Architecture – Architecting Your Web Store*, and you also saw how `InternalResourceViewResolver` resolves a particular logical View name into a View. Anyhow, I will repeat it briefly here.

`InternalResourceViewResolver` will resolve the actual View file path by prepending the configured `prefix` value and appending the `suffix` value with the logical View name;

the logical View name is the value usually returned by the Controller's method. So the Controller's method didn't return any actual View, it just returns the View name. It is the role of `InternalResourceViewResolver` to form the correct URL path for the actual `InternalResourceView`.

RedirectView

In a web application, URL redirection or forwarding is the technique of moving visitors to a different web page than the one they requested. Most of the time, this technique is used after submitting a web form to avoid resubmission of the same form due to pressing the browser's back button or refresh. Spring MVC has a special View object that handles redirection and forwarding. To use a `RedirectView` (`org.springframework.web.servlet.view.RedirectView`) with our Controller, we simply need to return the target URL string with the redirection prefix from the Controller. There are two redirection prefixes available in Spring MVC:

- `redirect`
- `forward`

Time for action – examining RedirectView

Though both redirection and forwarding are used to present a different web page than the one requested, there is a small difference between them. Let's try to understand them by examining them:

1. Open our `HomeController` class and add one more request mapping method as follows:

```
@RequestMapping("/welcome/greeting")
public String greeting() {
    return "welcome";
}
```

2. Now alter the return statement of the existing `welcome` request mapping method as follows and save it:

```
return "forward:/welcome/greeting";
```

3. Now run our application and enter the URL `http://localhost:8080/webstore/`. You will be able to see the welcome message on the web page.

4. Now again alter the return statement of the existing `welcome` request mapping method as follows and save it:

```
return "redirect:/welcome/greeting";
```

5. Now run our application and enter the URL `http://localhost:8080/webstore/`. You will see a blank page without any welcome message.
6. Revert the return value of the `welcome` method to the original value:

```
return "welcome";
```

What just happened?

What we demonstrated here is how to invoke `RedirectView` from the Controller's method. In step 1, we simply created a request mapping method called `greeting` for the `welcome/greeting` request path. This method simply returns a logical View name as `welcome`.

Since we returned the logical View name as `welcome`, the `welcome.jsp` file will be rendered by `InternalResourceView` at runtime. The `InternalResourceView` file expects two model attributes, `greeting` and `tagline`, while rendering `welcome.jsp`. And in step 2, we altered the return statement of the existing request mapping method to return a redirect URL as follows:

```
@RequestMapping("/")
public String welcome(Model model) {
    model.addAttribute("greeting", "Welcome to Web Store!");
    model.addAttribute("tagline", "The one and only amazing web store");
    return "forward:/welcome/greeting";
}
```

So what we did in step 2 was more important; instead of returning a logical View name, we simply returned the request path value of the `greeting` handler method with the `forward` keyword prefixed.

The moment Spring MVC sees this, it can understand that it is not a regular logical View name, so it won't search for any View file under the `src/main/webapp/WEB-INF/views/` directory; rather it will consider this request for forwarding to another request mapping method based on the request path attached after the `forward:` keyword.

One important thing to remember here is that the forwarded request is still the active original request, so whatever value we put in the model at the start of the request would still be available; that's why we did not add any value to the `Model` inside the `greeting` method. We simply returned the `View` name as `welcome` and the `welcome.jsp` file with the assumption that there will be model attributes, `greeting` and `tagline`, available in the model. So when we finally run our application as mentioned in step 3, even though we issued the request to the URL `http://localhost:8080/webstore/`, `RedirectView` will forward our request to `http://localhost:8080/webstore/welcome/greeting` and we will be able to see the welcome message on the web page.

Again in step 4, we simply changed the return statement of the `processAddNewProductFormwelcome` method to the `redirect: prefix`. This time Spring will consider this request as a new request, so whatever value we put in the model (inside the `welcome` method) at the start of the original request will be gone. This is why you saw an empty welcome page in step 6, since the `welcome.jsp` page can't read the `greeting` and `tagline` model attributes from the model.

So, based on this exercise, we understand that `RedirectView` will come into the picture if we return a redirection URL with the appropriate prefix from the Controller method. `RedirectView` will keep the original request or spawn a new request based on redirection or forwarding.

Pop quiz – RedirectView

Consider the following customer Controller:

```
@Controller("/customers")
public class CustomerController {
    @RequestMapping("/list")
    public String list(Model model) {
        return "customers";
    }

    @RequestMapping("/process")
    public String process(Model model) {
        // return
    }
}
```

If I want to redirect the `list` method from `process`, how should I form the return statement with the `process` method?

1. `return "redirect:list"`
2. `return "redirect:/list"`
3. `return "redirect:customers/list"`
4. `return "redirect:/customers/list"`

Flash attribute

In a normal Spring MVC application, every form submitted POSTs the form data to the server; a normal Spring Controller retrieves the data from those forms from the request and processes it further. Once the operation is successful, the user is forwarded to another page showing a message that the operation was a success.

Traditionally, if we handle this scenario via the `POST/Forward/GET` pattern, then it may sometimes cause multiple form submission issues. The user might press *F5* and the same form will be submitted again. To resolve this issue, the `POST/Redirect/GET` pattern is used in many web applications. Once the user's form is submitted successfully, we redirect the request to another success page instead of forwarding it. This makes the browser perform a new GET request and load the GET page. Thus if the user even presses *F5* multiple times, the GET request gets loaded instead of submitting the form again and again.

While the `POST/Redirect/GET` pattern seems to perfectly solve the problem of multiple form submissions, it adds one more problem of retrieving request parameters and attributes from the initial POST request. Usually, when we perform an HTTP request redirection, the data stored in the original request is lost, making it impossible for the next GET request to access it after redirection. Flash attributes can help in such cases. Flash attributes provide a way for us to store information that is intended to be used in another request. Flash attributes are saved temporarily in a session to be available for an immediate request after redirection.

In order to use Flash attributes in your Spring MVC application, just add the `RedirectAttributes redirectAttributes` parameter to your Spring Controller's method as follows:

```
@RequestMapping
public String welcome(Model model, RedirectAttributes redirectAttributes) {
    model.addAttribute("greeting", "Welcome to Web Store!");
    model.addAttribute("tagline", "The one and only amazing web store");
}
```

```
redirectAttributes.addFlashAttribute("greeting", "Welcome to Web Store!");
redirectAttributes.addFlashAttribute("tagline", "The one and only amazing
web store");
return "redirect:/welcome/greeting";
}
```

In this example, we stored two attributes, namely `greeting` and `tagline`, in Flash attributes; thus when the redirection happens to the `/welcome/greeting` (the `greeting` controller method in `HomeController`) request mapping method, it will have access to those variable values that are already in the model object.

Serving static resources

So far, we have seen that every request goes through the Controller and returns a corresponding View file for the request, and most of the time these View files contain dynamic content. By dynamic content, I mean that, during the request processing, the model values are dynamically populated in the View file. For example, if the View file is of the type JSP, then we populate the model values in the JSP file using the JSTL notation `${}`.

But what if we have some static content that we want to serve to the client? For example, consider an image that is static content; we don't want to go through Controllers in order to serve (fetch) an image, as there is nothing to process or update in terms of values in the model—we simply need to return the requested image.

Let's say we have a directory (`/resources/images/`) that contains some product images and we want to serve those images upon request. For example, if the requested URL is `http://localhost:8080/webstore/img/P1234.png`, then we will like to serve the image with the name `P1234.png`. Similarly, if the requested URL is `http://localhost:8080/webstore/img/P1236.png`, then an image with the name `P1236.png` needs to be served.

Time for action – serving static resources

Let's see how to serve static images with Spring MVC:

1. Put some images under the directory `src/main/webapp/resources/images/`;
I put three product images in: `P1234.png`, `P1235.png`, and `P1236.png`.

2. Override the `addResourceHandlers` method from our `WebApplicationContextConfig.java` web application context configuration file as follows:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry
registry) {
    registry.addResourceHandler("/img/**")
        .addResourceLocations("/resources/images/");
}
```

3. Now run our application and enter the URL `http://localhost:8080/webstore/img/P1234.png` (change the image name in the URL based on the images you put in the directory in step 1).
4. You will be able to view the image you requested in the browser.

What just happened?

What just happened was simple. In step 1, we put some image files under the `src/main/webapp/resources/images/` directory. And in step 2, we just overrode the `addResourceHandlers` method from our web application context configuration file, `WebApplicationContextConfig.java`, to tell Spring where those image files are located in our project, so that Spring can serve those files upon request:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/img/**")
        .addResourceLocations("/resources/images/");
}
```

The `addResourceLocations` method from `ResourceHandlerRegistry` defines the base directory location of the static resources that you want to serve. In our case, we want to serve all the images that are available under the `src/main/webapp/resources/images/` directory; you may wonder then why we have only given `/resources/images` as the location value instead of `src/main/webapp/resources/images/`. This is because, during our application build and deployment time, Spring MVC will copy everything available under the `src/main/webapp/` directory to the root directory of our web application. So during resource look up, Spring MVC will start looking up from the root directory.

The other method—`addResourceHandler`—just indicated the request path that needs to be mapped to this resource directory. In our case, we assigned `/img/**` as the mapping value. So if any web request comes with the request path `/img`, then it will be mapped to the `resources/images` directory, and the `**` symbol indicates to recursively look for any resource files underneath the base resource directory.

That is why, if you noticed in step 3, we formed the URL as follows:

`http://localhost:8080/webstore/img/P1234.png`. So while serving this web request, Spring MVC will consider `/img/P1234.png` as the request path, so it will try to map `/img` to the resource base directory `resources/images`. From that directory, it will try to look for the remaining path of the URL, which is `/P1234.png`. Since we have the `images` directory under the `resources` directory, Spring can easily locate the image file from the `images` directory.

So in our application, if any request comes with the request path prefix `/img` in its URL, then Spring will look into the `location` directory that is configured in `ResourceHandlerRegistry` and will return the requested file to the browser. Remember Spring allows you to not only host images, but also any type of static files such as PDFs, Word documents, Excel sheets, and so in this fashion.

It is good that we are able to serve product images without adding any extra request mapping methods in our Controller.

Pop quiz – static view

Consider the following resource configuration:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/resources/**")
        .addResourceLocations("/pdf/");
}
```

Under the `pdf` directory, if I have a sub-directory such as `product/manuals/`, which contains a PDF file called `manual-P1234.pdf`, how can I form the request path to access that PDF file?

1. `/pdf/product/manuals/manual-P1234.pdf`
2. `/resources/product/manuals/manual-P1234.pdf`
3. `/product/manuals/manual-P1234.pdf`
4. `/resource/pdf/product/manuals/manual-P1234.pdf`

Time for action – adding images to the product detail page

Let's extend this technique to show the product images in our product listing page and in the product detail page. Perform the following steps:

1. Open `products.jsp`, which you can find under the `/src/main/webapp/WEB-INF/views/` directory in your project, and add the following `` tag after the `<div class="thumbnail">` tag:

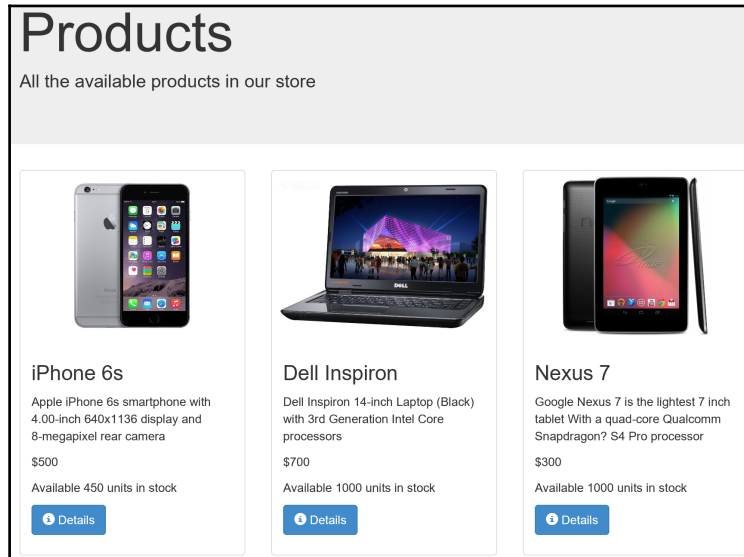
```

</c:url>" alt="image" style = "width:100%"/>
```

2. Similarly open `product.jsp` and add the following `` tag after the `<div class="row">` tag:

```
<div class="col-md-5">
  
  </c:url>" alt="image" style = "width:100%"/>
</div>
```

3. Now run our application and enter `http://localhost:8080/webstore/market/products`. You will be able to see a product list page with every product displaying a product image, as shown in the following screenshot:



Product listings with images

- Now click on the **Details** button of any product, and you will be able to see the corresponding View of the product details with an image attached to the details page, as follows:



Product detail page with image attached

What just happened?

What we did was simple. You learned how we can serve static resources, and you saw how to host product images. During this exercise, you learned that, in our application, if any request comes with the request path prefix `/img`, it will get mapped to the base resource directory and any further remaining URL path will lead to the static file.

We leveraged this fact and formed the image's `src` URL accordingly; notice the `src` attribute of the `` tag we added in step 1:

```
</c:url>"
alt="image" style = "width:100%"/>
```

The `src` attribute value that we are forming in this `` tag has an expression to fetch the product ID. After getting the product ID, we simply concatenate it to the existing value to form a valid request path, as follows:

```
"/img/${product.productId}.png"
```

For example, if the product ID is `P1234`, then we will get an image request URL as `/img/P1234.png`, which is nothing but one of the image file names that we already put in the `/resources/images` directory. So Spring can easily return that image file, which we showed using the `` tag in steps 1 and 2.

Multipart requests in action

In the preceding exercise, you saw how to incorporate a static View to show product images on the product details page. We simply put some images in a directory on the server and did some configuration, and Spring MVC was able to pick up those files while rendering the product details page. What if we automated this process? I mean, instead of putting those images in the directory, what if we were able to upload the images to the image directory?

How can we do this? Here comes the multipart request. A multipart request is a type of HTTP request to send files and data to the server. Spring MVC provides good support for multipart requests. Let's say we want to upload some files to the server, then we have to form a multipart request to accomplish that.

Time for action – adding images to a product

Let's add an image upload facility in our `add_products` page:

1. Add a bean definition in our web application context configuration file (`WebApplicationContextConfig.java`) for `CommonsMultipartResolver` as follows:

```
@Bean
public CommonsMultipartResolver multipartResolver() {
    CommonsMultipartResolver resolver=new
CommonsMultipartResolver();
    resolver.setDefaultEncoding("utf-8");
    return resolver;
}
```

2. Open `pom.xml`, which you can find under the project root directory itself.
3. You will be able to see some tabs under `pom.xml`; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
4. A **Select Dependency** window will appear; in **Group Id** enter `commons-fileupload`, in **Artifact Id** enter `commons-fileupload`, in **Version** enter `1.2.2`, select **Scope** as **compile**, then click on the **OK** button.
5. Similarly, add one more dependency: `org.apache.commons` as **Group Id**, `commons-io` as **Artifact Id**, `1.3.2` as **Version**, and **Scope** as **compile**, then click on the **OK** button and save `pom.xml`.
6. Open our product's domain class (`product.java`) and add a reference to `org.springframework.web.multipart.MultipartFile` with corresponding setters and getters as follows (don't forget to add getters and setters for this field):

```
private MultipartFile productImage;
```

7. Open `addProduct.jsp`, which you can find under the `/src/main/webapp/WEB-INF/views/` directory in your project, and add the following set of tags after the `<form:input id="condition">` tag group:

```
<div class="form-group">
    <label class="control-label col-lg-2" for="productImage">
        <spring:message code="addProduct.form.productImage.label"/>
    </label>
    <div class="col-lg-10">
        <form:input id="productImage" path="productImage"
```

```
type="file" class="form:input-large" />
</div>
</div>
```

8. Add an entry in our message bundle source (`messages.properties`) for the product's image label, as follows:

```
addProduct.form.productImage.label = Product Image file
```

9. Now set the `enctype` attribute to `multipart/form-data` in the form tag as follows and save `addProduct.jsp`:

```
<form:form modelAttribute="newProduct" class="form-
horizontal" enctype="multipart/form-data">
```

10. Open our `ProductController.java` and modify the `processAddNewProductForm` method's signature by adding an extra method parameter of the type `HttpServletRequest` (`javax.servlet.http.HttpServletRequest`); so basically your `processAddNewProductForm` method signature should look like the following code snippet:

```
public String processAddNewProductForm(
    @ModelAttribute("newProduct") Product newProduct,
    BindingResult result, HttpServletRequest request) {
```

11. Add the following code snippet inside the `processAddNewProductForm` method just before `productService.addProduct(newProduct)`:

```
MultipartFile productImage = newProduct.getProductImage();
String rootDirectory =
request.getSession().getServletContext().getRealPath("/");
    if (productImage!=null && !productImage.isEmpty()) {
        try {
            productImage.transferTo(new
File(rootDirectory+"resources\\images"+
newProduct.getId() + ".png"));
        } catch (Exception e) {
            throw new RuntimeException("Product Image saving
failed", e);
        }
    }
```

12. Within the `initialiseBinder` method, add a `productImage` field to the whitelisting set as follows:

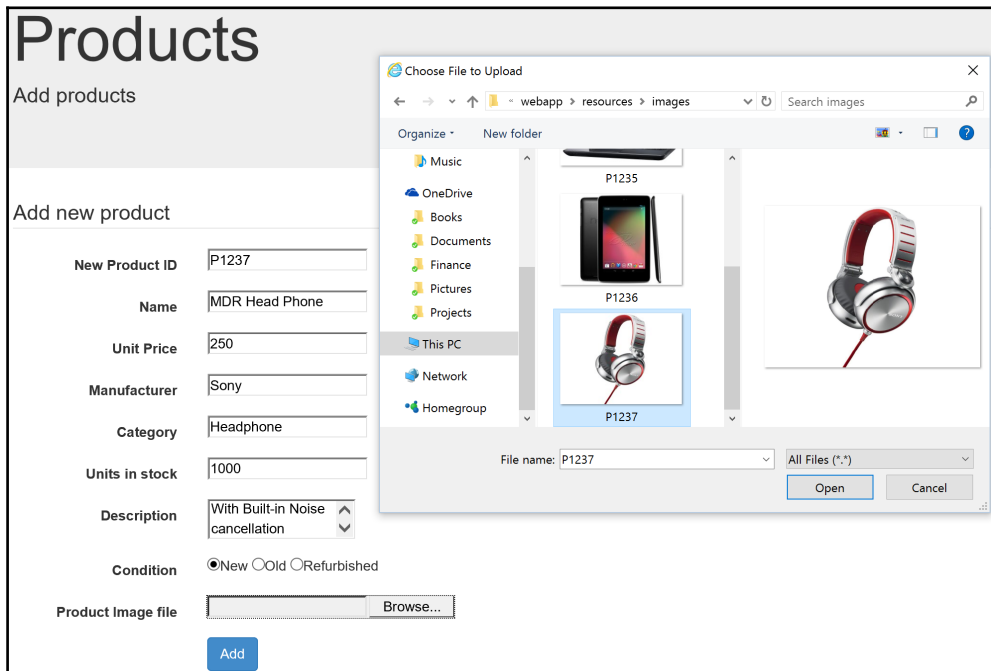
```

binder.setAllowedFields("productId",
    "name",
    "unitPrice",
    "description",
    "manufacturer",
    "category",
    "unitsInStock",
    "condition",
    "productImage");

```

13. Now run our application and enter the URL

<http://localhost:8080/webstore/market/products/add>. You will be able to see our add products page with an extra input field so you can choose which file to upload. Just fill out all the information as usual and, importantly, pick an image file of your choice for the newly-added image file; click on the **Add** button. You will be able to see that the image has been added to the **Products** page and to the product details page.



The add products page with the image selection option

What just happened?

Spring's `CommonsMultipartResolver`

(`org.springframework.web.multipart.commons.CommonsMultipartResolver`) class is the thing that determines whether the given request contains multipart content and parses the given HTTP request into multipart files and parameters. That's the reason we created a bean for that class within our web application context in step 1. And, through the `setMaxUploadSize` property, we set a maximum of 10,240,000 bytes as the allowed file size to be uploaded:

```
@Bean
public CommonsMultipartResolver multipartResolver() {
    CommonsMultipartResolver resolver=new CommonsMultipartResolver();
    resolver.setDefaultEncoding("utf-8");
    resolver.setMaxUploadSize(10240000);
    return resolver;
}
```

From steps 2 to 5, we added some of the `org.apache.commons` libraries as our Maven dependencies. This is because Spring uses those libraries internally to support the file uploading feature.

Since the image that we were uploading belongs to a product, it is better to keep that image as part of the product information; that's why in step 6 we added a reference to the `MultipartFile` in our domain class (`Product.java`) and added corresponding setters and getters. This `MultipartFile` reference holds the actual product image file that we are uploading.

We want to incorporate the image uploading facility in our `add products` page; that's why, in the `addProduct.jsp` View file, we added a file input tag to choose the desired image:

```
<div class="form-group">
<label class="control-label col-lg-2" for="productImage"> <spring:message
code="addProduct.form.productImage.label"/>
</label>
    <div class="col-lg-10">
        <form:input id="productImage" path="productImage" type="file"
class="form:input-large" />
    </div>
</div>
```

In the preceding set of tags, the important one is the `<form:input>` tag, which has the `type` attribute as `file` so that it can make the **Choose File** button display the file chooser window. As usual, we want this `form` field to be bound with the domain object field; that's

the reason we gave the `path` attribute as `productImage`. If you remember, this path name is just the same `MultipartFile` reference name that we added in step 6.

As usual, we want to externalize the label message for this file input tag as well; that's why we added `<spring:message>`, and in step 8 we added the corresponding message entry in the message source file (`messages.properties`).

Since our add product form is now capable of sending an image file as well as part of the request, we need to encode the request as a multipart request. This is why in step 9 we added the `enctype` attribute to the `<form:form>` tag and set its value as `multipart/form-data`. The `enctype` attribute indicates how the form data should be encoded when submitting it to the server.

We wanted to save the image file in the server under the `resources/images` directory, as this directory structure will be available directly under the root directory of our web application at runtime. So, in order to get the root directory of our web application, we need `HttpServletRequest`. See the following code snippet:

```
String rootDirectory =
    request.getSession().getServletContext().getRealPath("/");
```

That's the reason we added an extra method parameter called `request` of the type `HttpServletRequest` to our `processAddNewProductForm` method in step 10. Remember, Spring will fill this `request` parameter with the actual HTTP request.

In step 11, we simply read the image file from the domain object and wrote it into a new file with the product ID as the name:

```
MultipartFile productImage = newProduct.getProductImage();
String rootDirectory =
    request.getSession().getServletContext().getRealPath("/");
    if (!productImage.isEmpty()) {
        try {
            productImage.transferTo(new
File(rootDirectory+"resources\\images"+newProduct.getProductId() +
".png"));
        } catch (Exception e) {
            throw new RuntimeException("Product Image saving failed", e);
        }
    }
```

Remember, we purposely saved the images with the product ID name because we have already designed our products (`products.jsp`) page and details (`product.jsp`) page accordingly to show the right image based on the product ID.

And as a final step, we added the newly introduced `productImage` file to the whitelisting set in the binder configuration within the `initialiseBinder` method.

Now if you run your application and enter

`http://localhost:8080/webstore/market/products/add`, you will be able to see your `add products` page with an extra input field to choose the file to upload.

Have a go hero – uploading product user manuals to the server

It's nice that we were able to upload the product image to the server while adding a new product. Why don't you extend this facility to upload a PDF file to the server? For example, consider that every product has a user manual and you want to upload these user manuals while adding a product.

Here are some of the things you can do to upload PDF files:

- Create a directory with the name `pdf` under the `src/main/webapp/resources/` directory in your project
- Add one more `MultipartFile` reference in your product domain class (`Product.java`) to hold the PDF file and change `Product.java` accordingly
- Extend `addProduct.jsp`
- Extend `ProductController.java` accordingly; don't forget to add the newly added field to the whitelist

So finally, if the newly added product ID is `P1237`, you will be able to access the PDF under `http://localhost:8080/webstore/pdf/P1237.pdf`.

Good luck!

Using ContentNegotiatingViewResolver

Content negotiation is a mechanism that makes it possible to serve different representations of the same resource. For example, so far we have shown our product detail page in a JSP representation. What if we want to represent the same content in an XML format. Similarly, what if we want the same content in a JSON format? Here comes Spring MVC's `ContentNegotiatingViewResolver` (`org.springframework.web.servlet.view.ContentNegotiatingViewResolver`) to help us.

The XML and JSON formats are popular data interchange formats that are heavily used in web service communications. Using `ContentNegotiatingViewResolver`, we can incorporate many Views such as `MappingJacksonJsonView` (for JSON) and `MarshallingView` (for XML) to represent the same product information in a XML or JSON format.

Time for action – configuring `ContentNegotiatingViewResolver`

`ContentNegotiatingViewResolver` does not resolve Views itself, but rather delegates to other view resolvers based on the request. Now, let's add a content negotiation capability to our application:

1. Open `pom.xml`, which you can find under the project root directory.
2. You should be able to see some tabs under `pom.xml`; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; in **Group Id** enter `org.springframework`, in **Artifact Id** enter `spring-oxm`, in **Version** enter `4.3.0.RELEASE`, select **Scope** as **compile**, then click on the **OK** button.
4. Add another dependency: `org.codehaus.jackson` as **Group Id**, `jackson-mapper-asl` as **Artifact Id**, `1.9.10` as **Version**, and select **Scope** as **compile**. Then click on the **OK** button.
5. Similarly, add one more dependency: `com.fasterxml.jackson.core` as **Group Id**, `jackson-databind` as **Artifact Id**, `2.8.0` as **Version**, and select **Scope** as **compile**. Then click on the **OK** button and save `pom.xml`.
6. Now add the bean configuration in our web application context configuration (`WebApplicationContextConfig.java`) for the JSON View as follows:

```
@Bean
public MappingJackson2JsonView jsonView() {
    MappingJackson2JsonView jsonView = new
    MappingJackson2JsonView();
    jsonView.setPrettyPrint(true);
    return jsonView;
}
```

7. Add another bean configuration for the XML View as follows:

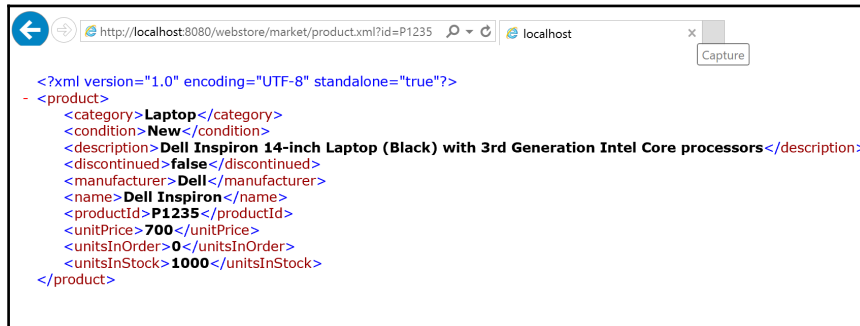
```
@Bean
public MarshallingView xmlView() {
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
}
```

```
        marshaller.setClassesToBeBound(Product.class);
        MarshallingView xmlView = new MarshallingView(marshaller);
        return xmlView;
    }
}
```

8. Finally, add the bean configuration for `ContentNegotiatingViewResolver` in our `WebApplicationContextConfig` web application context configuration file as follows:

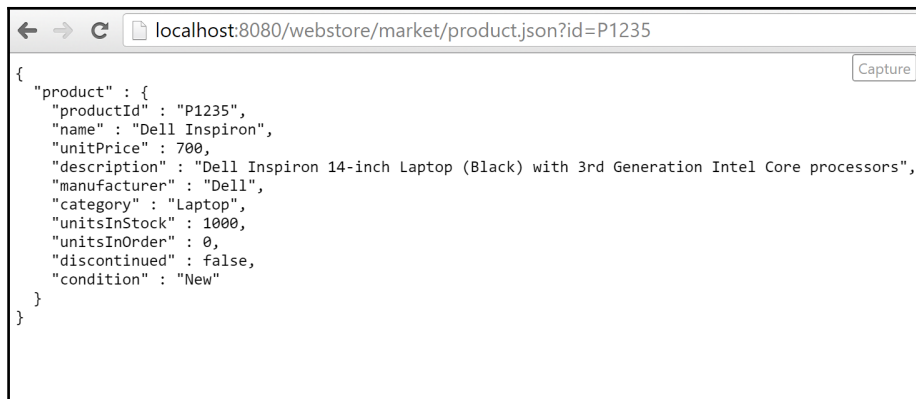
```
@Bean
public ViewResolver contentNegotiatingViewResolver(
    ContentNegotiationManager manager) {
    ContentNegotiatingViewResolver resolver = new
    ContentNegotiatingViewResolver();
    resolver.setContentNegotiationManager(manager);
    ArrayList<View> views = new ArrayList<>();
    views.add(jsonView());
    views.add(xmlView());
    resolver.setDefaultViews(views);
    return resolver;
}
```

9. Open our product domain class (`Product.java`) and add the `@XmlRootElement` annotation at the top of the class.
10. Similarly, add the `@XmlTransient` annotation at the top of the `getProductImage()` method and add another annotation `@JsonIgnore` on top of the `productImage` field.
11. Now run our application and enter the URL `http://localhost:8080/webstore/market/product?id=P1235`. You will now be able to view the details page of the product with the ID P1234.
12. Now change the URL to the `.xml` extension:
`http://localhost:8080/webstore/market/product.xml?id=P1235`. You should be able to see the same content in the XML format as shown in the following screenshot:



The product details page showing product information in the XML format

13. Similarly, this time change the URL to the `.json` extension:
`http://localhost:8080/webstore/market/product.json?id=P1235`. You will be able to see the JSON representation of that content as shown in the following screenshot:



The product details page showing the product information in the JSON format

What just happened?

Since we want an XML representation of our model data to convert our model objects into XML, we need Spring's object/XML mapping support—that's why we added the dependency for `spring-oxm.jar` through steps 1 to 3. The `spring-oxm` notation will help us convert an XML document to and from a Java object.

Similarly, to convert model objects into JSON, Spring MVC will use `jackson-mapper-asl.jar` and `jackson-databind.jar`; thus we need those jars in our project as well. In steps 4 and 5, we just added the dependency configuration for those jars.

If you remember in our servlet context (`servlet-context.xml`), we already defined `InternalResourceViewResolver` as our view resolver to resolve JSP-based Views, but this time we want a view resolver to resolve XML and JSON Views. That's why in step 8 we configured `ContentNegotiatingViewResolver` (`org.springframework.web.servlet.view.ContentNegotiatingViewResolver`) in our servlet context.

As I already mentioned, `ContentNegotiatingViewResolver` does not resolve Views itself but rather it delegates to other Views based on the request, so we need to introduce other Views to `ContentNegotiatingViewResolver`. How we do that is through the `setDefaultViews` method in `ContentNegotiatingViewResolver`:

```
@Bean
public ViewResolver contentNegotiatingViewResolver(
    ContentNegotiationManager manager) {
    ContentNegotiatingViewResolver resolver = new
ContentNegotiatingViewResolver();
    resolver.setContentNegotiationManager(manager);
    ArrayList<View> views = new ArrayList<>();
    views.add(jsonView());
    views.add(xmlView());
    resolver.setDefaultViews(views);
    return resolver;
}
```

To configure bean references for `jsonView` and `xmlView` inside `ContentNegotiatingViewResolver`, we need a bean definition for those references. That is the reason we defined those beans in steps 6 and 7.

The `xmlView` bean configuration especially has one important property to be set called `classesToBeBound`; this lists the domain objects that require XML conversion during the request processing. Since our product domain object requires XML conversion, we added `com.packt.webstore.domain.Product` to the list `classesToBeBound`:

```
@Bean
public MarshallingView xmlView() {
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
    marshaller.setClassesToBeBound(Product.class);
    MarshallingView xmlView = new MarshallingView(marshaller);
    return xmlView;
}
```

In order to convert it to XML, we need to give `MarshallingView` one more hint to identify the root XML element in the `Product` domain object. This is why in step 9 we annotated our class with the `@XmlRootElement` (`javax.xml.bind.annotation.XmlRootElement`) annotation.

In step 10, we added the `@XmlTransient` (`javax.xml.bind.annotation.XmlTransient`) annotation on top of the `getProductImage()` method and added another annotation—`@JsonIgnore` (`org.codehaus.jackson.annotate.JsonIgnore`)—on top of the `productImage` field. This is because we don't want to represent the product image as part of the XML View or the JSON View since both formats are purely a text-based representation, and it is not possible to represent images in text.

In step 10, we simply accessed our product details page in the regular way by firing the web request (`http://localhost:8080/webstore/products/product?id=P1234`) from the browser. We could see the normal JSP View, as expected.

In step 11, we just changed the URL slightly by adding an `.xml` extension to the request path:

`http://localhost:8080/webstore/market/products/product.xml?id=P1235`. This time we were able to see the same product information in the XML format.

Similarly for the JSON View, we changed the extension for the `.json` path to `http://localhost:8080/webstore/market/products/product.json?id=P1235` and we were able to see the JSON representation of the same product information.

Working with HandlerExceptionResolver

Spring MVC provides several approaches to exception handling. In Spring, one of the main exception handling constructs is the `HandlerExceptionResolver` (`org.springframework.web.servlet.HandlerExceptionResolver`) interface. Any objects that implement this interface can resolve exceptions thrown during Controller mapping or execution. `HandlerExceptionResolver` implementers are typically registered as beans in the web application context.

Spring MVC creates two such `HandlerExceptionResolver` implementations by default to facilitate exception handling:

- `ResponseStatusExceptionHandler` is created to support the `@ResponseStatus` annotation

- `ExceptionHandlerExceptionHandlerResolver` is created to support the `@ExceptionHandler` annotation

Time for action – adding a `ResponseStatus` exception

We will look at them one by one. First, the `@ResponseStatus` (`org.springframework.web.bind.annotation.ResponseStatus`) annotation; in Chapter 3, *Control Your Store with Controllers* we created a request mapping method to show products by category under the URI template: `http://localhost:8080/webstore/market/products/{category}`. If no products were found under the given category, we showed an empty web page, which is not correct semantically as we should show a HTTP status error to indicate that no products exist under the given category. Let's see how to do that with the help of the `@ResponseStatus` annotation:

1. Create a class called `NoProductsFoundUnderCategoryException` under the `com.packt.webstore.exception` package in the `src/main/java` source folder. Now add the following code to it:

```
package com.packt.webstore.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation
    .ResponseStatus;

@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="No
products found under this category")
public class NoProductsFoundUnderCategoryException extends
    RuntimeException{

    private static final long serialVersionUID =
        3935230281455340039L;
}
```

2. Now open our `ProductController` class and modify the `getProductsByCategory` method as follows:

```
@RequestMapping("/products/{category}")
public String getProductsByCategory(Model model,
    @PathVariable("category") String category) {
    List<Product> products =
        productService.getProductsByCategory(category);

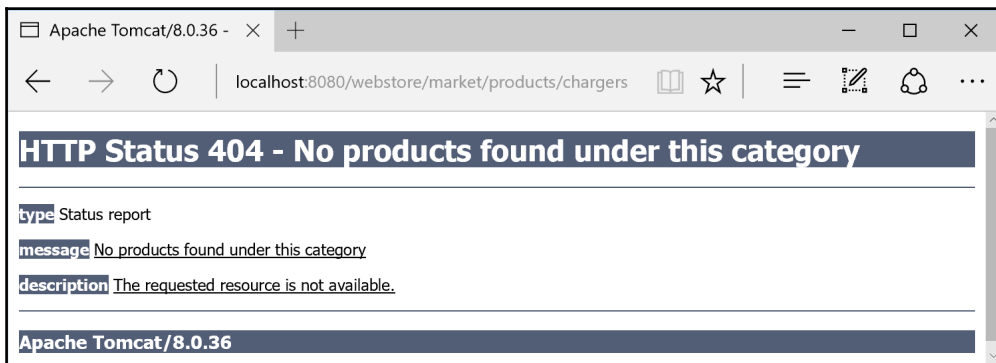
    if (products == null || products.isEmpty()) {
```

```
        throw new NoProductsFoundUnderCategoryException();
    }

    model.addAttribute("products", products);
    return "products";
}
```

3. Now run our application and enter the URL

`http://localhost:8080/webstore/market/products/chargers`. You will see a HTTP status error saying **No products found under this category**, as shown here:



Product category page showing the HTTP Status 404 for No products found under this category

What just happened?

In step 1, we created a runtime exception called

`NoProductsFoundUnderCategoryException` to indicate that no products were found under the given category. One of the important constructs in the

`NoProductsFoundUnderCategoryException` class is the `@ResponseStatus` annotation, which instructs the Spring MVC to return a specific HTTP status if this exception has been thrown from a request mapping method.

We can configure which HTTP status needs to be returned via the `value` attribute of the `@ResponseStatus` annotation; in our case we configured `HttpStatus.NOT_FOUND` (`org.springframework.http.HttpStatus`), which displays the familiar HTTP 404 response. The second attribute, `reason`, denotes the reason for the HTTP response error.

In step 2, we just modified the `getProductsByCategory` method in the `ProductController` class to check whether the product list for the given category is empty. If so, we simply throw the exception we created in step 1, which returns a **HTTP Status 404** error to the client saying **No products found under this category**.

So finally in step 3, we fired the web request `http://localhost:8080/webstore/market/products/chargers`, which would try to look for products under the category `chargers` but, since we didn't have any products under the `chargers` category, we got the **HTTP Status 404** error.

It's good that we can show the HTTP status error for products not found under a given category, but sometimes you may wish to have an error page where you want to show your error message in a more detailed manner.

For example, run our application and enter

`http://localhost:8080/webstore/market/product?id=P1234`. You will be able to see a detailed View of the iPhone 6s—now change the product ID in the URL to an invalid one such as `http://localhost:8080/webstore/market/product?id=P1000`, and you will see an error page.

Time for action – adding an exception handler

We should show a nice error message saying No products found with the given product ID, so let's do that with the help of `@ExceptionHandler`:

1. Create a class called `ProductNotFoundException` under the `com.packt.webstore.exception` package in the `src/main/java` source folder. Add the following code to it:

```
package com.packt.webstore.exception;

public class ProductNotFoundException extends
    RuntimeException{

    private static final long serialVersionUID =
        -694354952032299587L;
    private String productId;

    public ProductNotFoundException(String productId) {
        this.productId = productId;
    }

    public String getProductId() {
```

```
        return productId;
    }

}
```

2. Now open our `InMemoryProductRepository` class and modify the `getProductById` method as follows:

```
@Override
public Product getProductById(String productId) {
    String SQL = "SELECT * FROM PRODUCTS WHERE ID = :id";
    Map<String, Object> params = new HashMap<>();
    params.put("id", productId);
    try {
        return jdbcTemplate.queryForObject(SQL, params, new
ProductMapper());
    } catch (DataAccessException e) {
        throw new ProductNotFoundException(productId);
    }
}
```

3. Add an exception handler method using the `@ExceptionHandler` (`org.springframework.web.bind.annotation.ExceptionHandler`) annotation, as follows, in the `ProductController` class:

```
@ExceptionHandler(ProductNotFoundException.class)
public ModelAndView handleError(HttpServletRequest req,
ProductNotFoundException exception) {
    ModelAndView mav = new ModelAndView();
    mav.addObject("invalidProductId",
exception.getProductId());
    mav.addObject("exception", exception);
    mav.addObject("url",
req.getRequestURL()+"?" + req.getQueryString());
    mav.setViewName("productNotFound");
    return mav;
}
```

4. Finally, add one more JSP View file called `productNotFound.jsp` under the `src/main/webapp/WEB-INF/views/` directory, add the following code snippets to it, and save it:

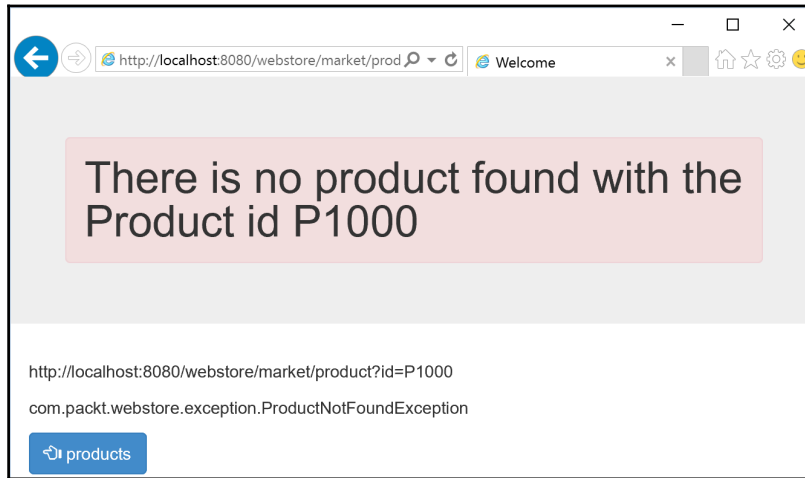
```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Welcome</title>
</head>
<body>
  <section>
    <div class="jumbotron">
      <div class="container">
        <h1 class="alert alert-danger"> There is no
        product found with the Product id
        ${invalidProductId}</h1>
      </div>
    </div>
  </section>

  <section>
    <div class="container">
      <p>${url}</p>
      <p>${exception}</p>
    </div>

    <div class="container">
      <p>
        <a href="<spring:url value="/market/products" />"
        class="btn btn-primary">
          <span class="glyphicon-hand-left glyphicon">
            </span> products
        </a>
      </p>
    </div>
  </section>
</body>
</html>
```

5. Now run our application and enter `http://localhost:8080/webstore/market/product?id=P1000`. You will see an error page showing **There is no product found with the Product id P1000** as follows:



Product detail page showing a custom error page for the product ID P1000

What just happened?

We decided to show a custom-made error page instead of showing the raw exception in the case of a product not being found for a given product ID. So, in order to achieve that, in step 1 we just created a runtime exception called `ProductNotFoundException` to be thrown when the product is not found for the given product ID.

In step 2, we just modified the `getProductById` method of the `InMemoryProductRepository` class to check whether any products were found for the given product ID. If not, we simply throw the exception (`ProductNotFoundException`) we created in step 1.

In step 3, we added our exception handler method to handle `ProductNotFoundException` with the help of the `@ExceptionHandler` annotation. Within the `handleError` method, we just created a `ModelAndView` (`org.springframework.web.servlet.ModelAndView`) object and stored the requested invalid product ID, exception, and the requested URL, and returned it with the View name `productNotFound`:

```
@ExceptionHandler(ProductNotFoundException.class)
public ModelAndView handleError(HttpServletRequest req,
ProductNotFoundException exception) {
    ModelAndView mav = new ModelAndView();
    mav.addObject("invalidProductId", exception.getProductId());
}
```

```
mav.addObject("exception", exception);
mav.addObject("url", req.getRequestURL()+"?" + req.getQueryString());
mav.setViewName("productNotFound");
return mav;
}
```

Since we returned the ModelAndView object with the View name `productNotFound`, we must have a View file with the name `productNotFound`. That's why we created this View file (`productNotFound.jsp`) in step 4. `productNotFound.jsp` just contains a CSS-styled `<h1>` tag to show the error message and a link button to the product listing page.

So, whenever we request to show a product with an invalid ID such as `http://localhost:8080/webstore/product?id=P1000`, the `ProductController` class will throw the `ProductNotFoundException`, which will be handled by the `handleError` method and will show the custom error page (`productNotFound.jsp`).

Summary

In this chapter, you learned how `InternalResourceViewResolver` resolves Views, and you saw how to activate `RedirectView` from a Controller's method. You learned the important difference between `redirect` and `forward` and also found out about Flash attributes. After that, you saw how to host static resources files without going through Controller configuration. You discovered how to attach a static image file to the product details page. You learned how to upload files to the server. You saw how to configure `ContentNegotiatingViewResolver` to give alternate XML and JSON Views for product domain objects in our application. Finally, you made use of `HandlerExceptionResolver` to resolve exceptions.

In the next chapter, you will learn how to intercept regular web requests with the help of an interceptor. See you in the next chapter.

6

Internalize Your Store with Interceptor

In all our previous chapters, we have only seen how to map a request to the Controller's method; once the request reaches the Controller method, we execute some logic and return a logical View name, which can be used by the view resolver to resolve Views, but what if we want to execute some logic before actual request processing happens? Similarly, what if we want to execute some other instruction before dispatching the response?

Spring MVC interceptor intercepts the actual request and response. Interceptors are a special web programming technique, where we can execute a certain piece of logic before or after a web request is processed. In this chapter, we are going to learn more about interceptors. After finishing this chapter, you will know:

- How to configure an interceptor
- How to add internalization support
- Data auditing using an interceptor
- Conditional redirecting using an interceptor

Working with interceptors

As I have already mentioned, interceptors are used to intercept actual web requests before or after processing them. We can relate the concept of interceptors in Spring MVC to the filter concept in Servlet programming. In Spring MVC, interceptors are special classes that must implement the `org.springframework.web.servlet.HandlerInterceptor` interface. The `HandlerInterceptor` interface defines three important methods, as follows:

- `preHandle`: This method will get called just before the web request reaches the Controller for execution
- `postHandle`: This method will get called just after the Controller method execution
- `afterCompletion`: This method will get called after the completion of the entire web request cycle

Once we create our own interceptor, by implementing the `HandlerInterceptor` interface, we need to configure it in our web application context in order for it to take effect.

Time for action – configuring an interceptor

Every web request takes a certain amount of time to get processed by the server. In order to find out how much time it takes to process a web request, we need to calculate the time difference between the starting time and ending time of a web request process. We can achieve that by using the interceptor concept. Let's see how to configure our own interceptor in our project to log the execution time of every web request:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project itself.
2. You will see some tabs at the bottom of the `pom.xml` file; select the **Dependencies** tab and click on the **Add** button in the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as `log4j`, **Artifact Id** as `log4j`, and **Version** as `1.2.17`, select **Scope** as **compile**, click the **OK** button, and save `pom.xml`.
4. Create a class named `ProcessingTimeLogInterceptor` under the `com.packt.webstore.interceptor` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.interceptor;

import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;

import org.apache.log4j.Logger;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class ProcessingTimeLogInterceptor implements
HandlerInterceptor {
    private static final Logger LOGGER =
Logger.getLogger(ProcessingTimeLogInterceptor.class);
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) {
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);

        return true;
    }
    public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler, ModelAndView
modelAndView) {
        String queryString = request.getQueryString() == null ?
"" : "?" + request.getQueryString();
        String path = request.getRequestURL() + queryString;

        long startTime = (Long)
request.getAttribute("startTime");
        long endTime = System.currentTimeMillis();
        LOGGER.info(String.format("%s millisecond taken to
process the request %s.", (endTime - startTime), path));
    }
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception
exceptionIfAny){
        // NO operation.
    }
}
```

5. Now open your web application context configuration file

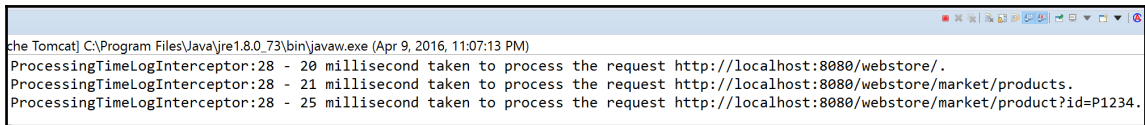
WebApplicationContextConfig.java, add the following method to it, and save the file:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new
ProcessingTimeLogInterceptor());
}
```

6. Create a property file named `log4j.properties` under the `src/main/resources` directory, and add the following content. Then, save the file:

```
# Root logger option
log4j.rootLogger=INFO, file, stdout
# Direct log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File= C:\\webstore\\webstore-
performance.log
log4j.appender.file.MaxFileSize=1MB
log4j.appender.file.MaxBackupIndex=1
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n
# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n
```

7. Now run our application, enter the `http://localhost:8080/webstore/market/products` URL, and navigate to some pages; you should be able to see the logging in the console as follows:



```
che Tomcat] C:\Program Files\Java\jre1.8.0_73\bin\javaw.exe (Apr 9, 2016, 11:07:13 PM)
ProcessingTimeLogInterceptor:28 - 20 millisecond taken to process the request http://localhost:8080/webstore/.
ProcessingTimeLogInterceptor:28 - 21 millisecond taken to process the request http://localhost:8080/webstore/market/products.
ProcessingTimeLogInterceptor:28 - 25 millisecond taken to process the request http://localhost:8080/webstore/market/product?id=P1234.
```

Showing ProcessingTimeLogInterceptor logging messages in the console

8. Open the file `C:\\webstore\\webstore-performance.log`; you can see the same log message in the logging file as well.

What just happened?

Our intention was to record the execution time of every request coming to our web application, so we decided to record the execution times in a log file. In order to use a logger, we needed the `log4j` library, so we added the `log4j` library as a Maven dependency in step 3.

In step 4, we defined an interceptor class named `ProcessingTimeLogInterceptor` by implementing the `HandlerInterceptor` interface. As we already saw, there are three methods that need to be implemented. We will see each method one by one. The first method is `preHandle()`, which is called before the execution of the Controller method:

```
public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) {
    long startTime = System.currentTimeMillis();
    request.setAttribute("startTime", startTime);
    return true;
}
```

In the previously shown `preHandle` method, we just set the current time value in the request object for later retrieval. Whenever a request comes to our web application, it first comes through this `preHandle` method and sets the current time in the request object before reaching the Controller. We are returning `true` from this method because we want the execution chain to proceed with the next interceptor or the Controller itself. Otherwise, `DispatcherServlet` assumes that this interceptor has already dealt with the response itself. So if we return `false` from the `preHandle` method, the request won't proceed to the Controller or the next interceptor.

The second method is `postHandle`, which will be called after the Controller method's execution:

```
public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) {
    String queryString = request.getQueryString() == null ? "" : "?" +
request.getQueryString();
    String path = request.getRequestURL() + queryString;

    long startTime = (Long) request.getAttribute("startTime");
    long endTime = System.currentTimeMillis();
    LOGGER.info(String.format("%s millisecond taken to process the request
%s.", (endTime - startTime), path));
}
```

In the preceding method, we are simply logging the difference between the current time, which is considered to be the request processing finish time, and the start time, which we got from the request object. Our final method is `afterCompletion`, which is called after the View is rendered. We don't want to put any logic in this method, that's why I left the method empty.



If you don't want to implement all the methods from the `HandlerInterceptor` interface in your interceptor class, you could consider extending your interceptor from `org.springframework.web.servlet.handler.HandlerInterceptorAdapter`, which is a convenient class provided by Spring MVC as a default implementation of all the methods from the `HandlerInterceptor` interface.

After creating our `ProcessingTimeLogInterceptor`, we need to register our interceptor with Spring MVC, which is what we have done in step 5 through the `addInterceptors` overridden method of `WebMvcConfigurerAdapter`:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new ProcessingTimeLogInterceptor());
}
```

In step 6, we have added a `log4j.properties` file in order to specify some logger-related configurations. You can see we have configured the log file location in `log4j.properties` as follows:

```
log4j.appender.file.File= C:\\webstore\\webstore-performance.log
```

Finally, in step 7 we ran our application in order to record some performance logging, and we were able to see that the logger is just working fine via the console. You can open the log file to view the performance logs.

So we understood how to configure an interceptor and saw `ProcessingTimeLogInterceptor` in action. In the next exercise, we will see how to use some Spring-provided interceptors.

Pop quiz – interceptors

Consider the following interceptor:

```
public class SecurityInterceptor extends HandlerInterceptorAdapter{

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex) throws
        Exception {
        // just some code related to after completion
    }
}
```

Is this `SecurityInterceptor` class a valid interceptor?

1. It is not valid because it does not implement the `HandlerInterceptor` interface.
2. It is valid because it extends the `HandlerInterceptorAdapter` class.

Within the interceptor methods, what is the order of execution?

1. `preHandle`, `afterCompletion`, `postHandle`.
2. `preHandle`, `postHandle`, `afterCompletion`.

LocaleChangeInterceptor – internationalization

In the previous sections, we saw how to create an interceptor (`ProcessingTimeLogInterceptor`) and configure it in our web application context. Spring provides some pre-built interceptors that we can configure in our application context as and when needed. One such pre-built interceptor is `LocaleChangeInterceptor`, which allows us to change the current locale on every request and configures `LocaleResolver` to support internationalization.

Internationalization means adapting computer software to different languages and regional differences. For example, if you are developing a web application for a Dutch-based company, they may expect all the web page text to be displayed in the Dutch language, use the Euro for currency calculations, expect a space as a thousand separator when displaying numbers, and use a “,” (comma) as a decimal point. On the other hand, when the same Dutch company wants to open a market in America, they expect the same web application to be adapted for American locales; for example, the web pages should be displayed in English, dollars should be used for currency calculations, numbers should be formatted with “,” (comma) as the thousand separator, a “.” (dot) acts as a decimal point, and so on.

The technique for designing a web application that can automatically adapt to different regions and countries without needing to be re-engineered is called internationalization, sometimes shortened to i18n (i-eighteen letters-n).

In Spring MVC, we can achieve internationalization through `LocaleChangeInterceptor` (`org.springframework.web.servlet.i18n.LocaleChangeInterceptor`). The `LocaleChangeInterceptor` allows us to change the current locale for every web request via a configurable request parameter.

In Chapter 4, *Working with Spring Tag Libraries*, we have seen how to externalize text messages in the add products page; now we are going to add internationalization support for the same add products page (`addProducts.jsp`), because in Spring MVC, prior to internationalizing a label, we must externalize that label first. Since we already externalized all the label messages in the add products page (`addProducts.jsp`), we can proceed to internationalize the add products page.

Time for action – adding internationalization

Technically we can add support for as many languages as we want with internationalization, but for demonstration purposes I am going to show you how to make our add products page with Dutch language support:

1. Create a file called `messages_nl.properties` under `/src/main/resources` in your project, add the following lines in it, and save the file:

```
addProduct.form.productId.label = Nieuw product ID
addProduct.form.name.label = Naam
addProduct.form.unitPrice.label = prijs per eenheid
addProduct.form.manufacturer.label = fabrikant
addProduct.form.category.label = categorie
addProduct.form.unitsInStock.label = Aantal op voorraad
addProduct.form.description.label = Beschrijving
addProduct.form.condition.label = Product Staat
addProduct.form.productImage.label = product afbeelding
```

2. Open our `addProduct.jsp` and add the following set of tags right after the `<body>` tag:

```
<section>
  <div class="pull-right" style="padding-right:50px">
    <a href="?language=en" >English</a>|<a href="?
      language=nl" >Dutch</a>
  </div>
</section>
```

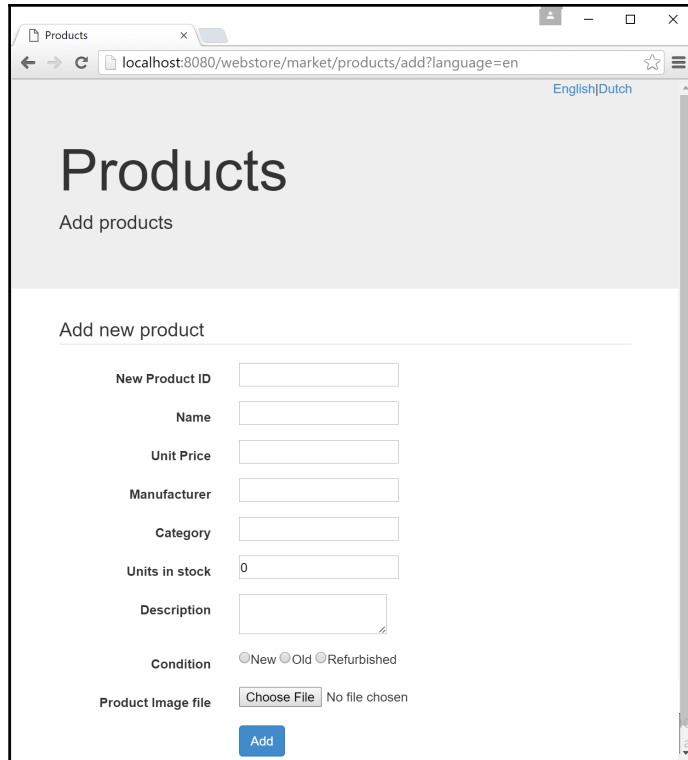
3. Now open our web application context configuration file, `WebApplicationContextConfig.java`, and add one more bean definition for the locale resolver as follows:

```
@Bean
public LocaleResolver localeResolver(){
    SessionLocaleResolver resolver = new
    SessionLocaleResolver();
    resolver.setDefaultLocale(new Locale("en"));
    return resolver;
}
```

4. Now change our `addInterceptors` method to configure one more interceptor in our `InterceptorRegistry` as follows:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new
    ProcessingTimeLogInterceptor());
    LocaleChangeInterceptor localeChangeInterceptor = new
LocaleChangeInterceptor();
localeChangeInterceptor.setParamName("language");
registry.addInterceptor(localeChangeInterceptor);
}
```


5. Now run our application and enter the `http://localhost:8080/webstore/market/products/add` URL; you will be able to see our regular add products page with two extra links in the top-right corner to choose the language:



The add products page showing internationalization support to choose languages

6. Now click on the **Dutch** link; you will see the product ID label transformed into the Dutch caption **Nieuw product ID**.
7. Since our configured `LocaleChangeInterceptor` will add a request parameter called `language` to the web request, we need to add this language request parameter to our whitelisting set in our `ProductController`. Open our `ProductController` and, within the `initialiseBinder` method, add the language request parameter to the whitelisting set as follows:

```
binder.setAllowedFields("productId",
```

```
"name",  
"unitPrice",  
"description",  
"manufacturer",  
"category",  
"unitsInStock",  
"condition",  
"productImage",  
"language");
```

What just happened?

In step 1, we just created a property file called `messages_nl.properties`. This file acts as a Dutch-based message source for all our externalized label messages in the `addProducts.jsp` file. In order to display the externalized label messages, we used the `<spring:message>` tag in our `addProducts.jsp` file.

But by default, the `<spring:message>` tag will read messages from the `messages.properties` file only; we need to make a provision for our end user to switch to the Dutch locale when they view the webpage, so that the label messages can come from the `messages_nl.properties` file. We provided such a provision through a locale-choosing link in `addProducts.jsp`, as mentioned in step 2:

```
<a href="?language=en" >English</a>|<a href="?language=nl" >Dutch</a>
```

In step 2, we created two links, one each to choose English or Dutch as the preferred locale. When the user clicks on one of these links, it will add a request parameter called `language` to the URL with the corresponding locale value. For example, when we click on the **English** link on the add products page at runtime, it will change the request URL to `http://localhost:8080/webstore/products/add?language=en`; similarly if you click on the **Dutch** link, it will change the request URL to `http://localhost:8080/webstore/products/add?language=nl`.

In step 3, we created a `SessionLocaleResolver` bean in our web application context as follows:

```
@Bean  
public LocaleResolver localeResolver(){  
    SessionLocaleResolver resolver = new SessionLocaleResolver();  
    resolver.setDefaultLocale(new Locale("en"));  
    return resolver;  
}
```

`SessionLocaleResolver` is the one that sets the locale attribute in the user's session. One important property of `SessionLocaleResolver` is `defaultLocale`. We assigned `en` as the value for the default locale, which indicates that by default our page should use English as its default locale.

In step 4, we created a `LocaleChangeInterceptor` bean and configured it in the existing interceptor list:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new ProcessingTimeLogInterceptor());
    LocaleChangeInterceptor localeChangeInterceptor = new
    LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("language");
    registry.addInterceptor(localeChangeInterceptor);
}
```

We assigned the string `language` as the value for the `paramName` property in `LocaleChangeInterceptor`. There is a reason for this because, if you notice, in step 2 when we created the locale choosing link in `add products` page (`addProduct.jsp`), we used the same parameter name as the request parameter within the `<a>` tag:

```
<a href="?language=en" >English</a>|<a href="?language=nl" >Dutch</a>
```

This way we can give a hint to `LocaleChangeInterceptor` to choose the correct user-preferred locale. So whatever parameter name you plan to use in your URL, use the same name as the value for the `paramName` property in `LocaleChangeInterceptor`. One more thing: whatever value you have given to the `language` request parameter in the link should match the translation message source file suffix. For example, in our case we have created a Dutch translation message source file, `messages_nl.properties`; here the suffix is `nl` and `messages.properties` without any suffix is considered the default for the `en` suffix. That's why in step 2 we have given `nl` and `en` as the values for the `language` parameters for Dutch and English, respectively:

```
<a href="?language=en" >English</a>|<a href="?language=nl" >Dutch</a>
```

So finally, when we run our application and enter the `http://localhost:8080/webstore/market/products/add` URL, you will see our regular `add products` page with two extra links in the top-right corner to choose the language.

Clicking on **Dutch** changes the request URL

to `http://localhost:8080/webstore/market/products/add?language=nl`, which brings up the `LocaleChangeInterceptor` and reads the Dutch-based label messages from `messages_nl.properties`.

Note that, if we do not give a language parameter in our URL, Spring will use the default message source file (`messages.properties`) for translation; if we give a language parameter, Spring will use that parameter value as the suffix to identify the correct language message source file (`messages_nl.properties`).

Have a go hero – fully internationalize the product details page

As I already mentioned just for demonstration purposes, I have internationalized a single web page (`addProducts.jsp`). I encourage you to internationalize the product detail web page (`product.jsp`) in our project. You can use the Google translate service (<https://translate.google.com/>) to find out the Dutch translation for labels. Along with this, try to add language support for one more language of your choice.

Mapped interceptors

Interceptors are a great way to add logic that needs to be executed during every request to our web application. The cool thing about interceptors is that, not only can we have a logic that executes before and/or after the request, we can even bypass or redirect the original web request itself. So far, we have seen a couple of examples of interceptors, such as performance logging and internationalization, but one problem with those examples is that there is no proper way to stop them from being run for specific requests.

For example, we might have a specific web request that does not need to run that additional logic in our interceptor. How can we tell Spring MVC to selectively include or exclude interceptors? Mapped interceptors to the rescue; using mapped interceptors we can selectively include or exclude interceptors, based on a mapped URL. Mapped interceptors use Ant-based path pattern matching to determine whether a web request path matches the given URL path pattern.

Consider a situation where you want to show the special-offer products page only to those users who have a valid promo code, but others who are trying to access the special-offer products page with an invalid promo code should be redirected to an error page.

Time for action – mapped intercepting offer page requests

Let's see how we can achieve this piece of functionality with the help of a mapped interceptor:

1. Create a class named `PromoCodeInterceptor` under the `com.packt.webstore.interceptor` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.interceptor;

import java.io.IOException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.handler
.HandlerInterceptorAdapter;

public class PromoCodeInterceptor extends
HandlerInterceptorAdapter {

    private String promoCode;
    private String errorRedirect;
    private String offerRedirect;

    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws
IOException{
        String givenPromoCode = request.getParameter("promo");

        if (promoCode.equals(givenPromoCode)) {
            response.sendRedirect(request.getContextPath() + "/"
+ offerRedirect);
        } else {
            response.sendRedirect(errorRedirect);
        }

        return false;
    }

    public void setPromoCode(String promoCode) {
        this.promoCode = promoCode;
    }

    public void setErrorRedirect(String errorRedirect) {
```

```
        this.errorRedirect = errorRedirect;
    }

    public void setOfferRedirect(String offerRedirect) {
        this.offerRedirect = offerRedirect;
    }
}
```

2. Create a bean definition for `PromoCodeInterceptor` in our web application context (`WebApplicationContextConfig.java`) as follows:

```
@Bean
public HandlerInterceptor promoCodeInterceptor() {
    PromoCodeInterceptor promoCodeInterceptor = new
    PromoCodeInterceptor();
    promoCodeInterceptor.setPromoCode("OFF3R");
    promoCodeInterceptor.setOfferRedirect("market/products");
    promoCodeInterceptor.setErrorRedirect("invalidPromoCode");
    return promoCodeInterceptor;
}
```

3. Now configure the `PromoCodeInterceptor` interceptor bean in our `InterceptorRegistry` as follows:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new
    ProcessingTimeLogInterceptor());
    LocaleChangeInterceptor localeChangeInterceptor = new
    LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("language");
    registry.addInterceptor(localeChangeInterceptor);
    registry.addInterceptor(promoCodeInterceptor())
    .addPathPatterns("/**/market/products/specialOffer");
}
```

4. Open our `ProductController` class and add one more request mapping method as follows:

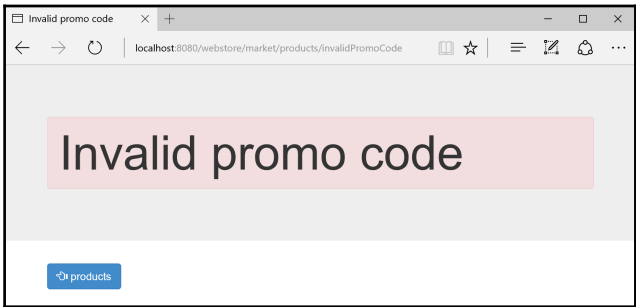
```
@RequestMapping("/products/invalidPromoCode")
public String invalidPromoCode() {
    return "invalidPromoCode";
}
```

5. Finally add one more JSP View file called `invalidPromoCode.jsp` under the directory `src/main/webapp/WEB-INF/views/`, add the following code snippets to it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet" href="//netdna.bootstrapcdn.com/
bootstrap/3.0.0/css/bootstrap.min.css">
<title>Invalid promo code</title>
</head>
<body>
<section>
<div class="jumbotron">
<div class="container">
<h1 class="alert alert-danger"> Invalid promo
code</h1>
</div>
</div>
</section>

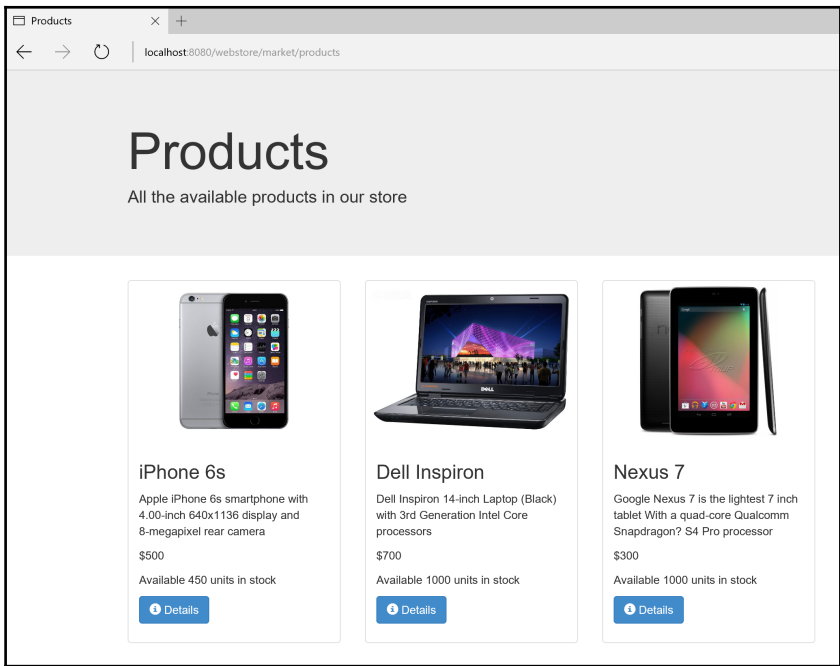
<section>
<div class="container">
<p>
<a href="<spring:url value="/market/products" />"
class="btn btn-primary">
<span class="glyphicon-hand-left glyphicon">
</span> products
</a>
</p>
</div>
</section>
</body>
</html>
```

6. Now run our application and enter the `http://localhost:8080/webstore/market/products/specialOffer?promo=offer` URL; you should be able to see an error message page as follows:



Showing invalid promo code exception

7. Now enter the `http://localhost:8080/webstore/market/products/specialOffer?promo=OFF3R` URL; you will land on a special-offer products page.



Showing special-offer page

What just happened?

The `PromoCodeInterceptor` class that we created in step 1 is very similar to `ProcessingTimeLogInterceptor`; the only difference is that we have extended `HandlerInterceptorAdapter` and only overridden the `preHandle` method:

```
public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws IOException{
    String givenPromoCode = request.getParameter("promo");

    if (promoCode.equals(givenPromoCode)) {
        response.sendRedirect(request.getContextPath() + "/" +
offerRedirect);
    } else {
        response.sendRedirect(errorRedirect);
    }

    return false;
}
```

In the `preHandle` method, we are simply checking whether the request contains the correct promo code as the HTTP parameter. If so, we redirect the request to the configured special-offer page; otherwise, we redirect it to the configured error page.

The `PromoCodeInterceptor` class has three properties, `promoCode`, `errorRedirect`, and `offerRedirect`. The `promoCode` property is used to configure the valid promo code; in our case, we have assigned `OFF3R` as the valid promo code in step 2, so whoever is accessing the special-offer page should provide `OFF3R` as the promo code in their HTTP parameter in order to access the special-offer page.

The next two attributes, `errorRedirect` and `offerRedirect`, are used for redirection. `errorRedirect` indicates the redirect URL mapping in the case of an invalid promo code and `offerRedirect` indicates the redirect URL mapping for a successful promo code redirection.



I have not created a special-offer products page; for demonstration purposes I have reused the same regular products page as a special-offer products page, which is why we have assigned `market/products` as the value for `offerRedirect`; thus, in the case of a valid promo code we are simply redirecting to the regular products page. But if we create a special-offer products page, we can assign that page URL as the value for `offerRedirect`.

Okay, we have created the `PromoCodeInterceptor` bean, but we have to configure this interceptor with our Spring MVC runtime, which is what we have done in step 3 by adding the `PromoCodeInterceptor` bean to our `InterceptorRegistry` within the `addInterceptors` method:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new ProcessingTimeLogInterceptor());
    LocaleChangeInterceptor localeChangeInterceptor = new
LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("language");
    registry.addInterceptor(localeChangeInterceptor);
    registry.addInterceptor(promoCodeInterceptor())
    .addPathPatterns("/**/market/products/specialOffer");
}
```

If you notice, while adding `promoCodeInterceptor` to our `InterceptorRegistry`, we can specify URL patterns using the `addPathPatterns` method. This way we can specify the URL patterns to which the registered interceptor should apply. So our `promoCodeInterceptor` will get executed only for a request that ends with `market/specialOffer`.



While adding an interceptor, we can also specify the URL patterns that the registered interceptor should not apply to via the `excludePathPatterns` method.

In step 4, we have added one more request mapping method called `invalidPromoCode` to show an error page in the case of an invalid promo code. In step 5, we have added the corresponding error View file, called `invalidPromoCode.jsp`.

So in step 6, we purposely entered the `http://localhost:8080/webstore/market/products/specialOffer?promo=offer` URL into our running application to demonstrate `PromoCodeInterceptor` in action; we saw the error page because the promo code we gave in the URL is `offer` (`?promo=offer`), which is wrong. In step 7, we gave the correct promo code in the `http://localhost:8080/webstore/market/products/specialOffer?promo=OFF3R` URL, so we should be able to see the configured special-offer products page.

Summary

In this chapter, we learned about the concept of interceptors and we learned how to configure interceptors in Spring MVC. We have seen how to do performance logging using interceptors. We also learned how to use Spring's `LocaleChangeInterceptor` to support internationalization. Later, we have seen how to do conditional redirecting using a mapped interceptor.

In the next chapter, I will introduce you to incorporating Spring Security into our web application. You will learn how to do authentication and authorization using the Spring Security framework.

7

Incorporating Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework for enterprise Java applications. The Spring Security framework is mainly used to ensure web application security such as authentication and authorization on application-level operations.

For web-layer security, Spring Security heavily leverages the existing Servlet filter architecture; it does not depend on any particular web technology. Spring Security mainly concerns `HttpServletRequest` and `HttpServletResponse` objects; it doesn't care about the source of the request and the response target. A request may originate from a web browser, web service, HTTP client, or JavaScript-based Ajax request. The only critical requirement for Spring Security is that it must be an `HttpServletRequest`, so that it can apply standard Servlet filters.

Spring Security provides various pre-built Servlet filters as part of the framework; the only requirement for us is to configure the appropriate filters in our web application in order to intercept the request and perform security checks. Spring Security is a vast topic, so we are not going to see all the capabilities of Spring Security; instead we are only going to see how to add basic authentication to our web pages.

Using Spring Security

In Chapter 4, *Working with Spring Tag Libraries*, we saw how to serve and process web forms; in that exercise we created a web page to add products. Anyone with access to the add products page could add new products to our web store. But in a typical web store, only administrators can add products. So how can we prevent other users from accessing the add products page? Spring Security comes to the rescue.

Time for action – authenticating users based on roles

We are going to restrict access to all our web pages using Spring Security. Only an authorised user or administrator with a valid username and password can access our web pages from a browser:

1. Open `pom.xml`; you can find `pom.xml` under the project root folder itself.
2. You should see some tabs at the bottom of `pom.xml`; select the **Dependencies** tab and click the **add** button in the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as `org.springframework.security`, **Artifact Id** as `spring-security-config`, **Version** as `4.1.1.RELEASE`, select **Scope** as **compile**, and click the **OK** button.
4. Similarly, add one more dependency **Group Id** as `org.springframework.security`, **Artifact Id** as `spring-security-web`, **Version** as `4.1.1.RELEASE`, select **Scope** as **compile**, and click the **OK** button. And most importantly, save `pom.xml`.
5. Now create one more controller class called `LoginController` under the `com.packt.webstore.controller` package in the `src/main/java` source folder. Add the following code to it:

```
package com.packt.webstore.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class LoginController {

    @RequestMapping(value = "/login", method =
        RequestMethod.GET)
    public String login() {
        return "login";
    }
}
```

6. Add one more JSP view file called `login.jsp` under the `src/main/webapp/WEB-INF/views/` directory, add the following code snippets into it, and save it:

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form"%>
```

```
<%@ taglib prefix="spring"
    uri="http://www.springframework.org/tags"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
    href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Products</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Welcome to Web Store!</h1>
                <p>The one and only amazing web store</p>
            </div>
        </div>
    </section>
    <div class="container">
        <div class="row">
            <div class="col-md-4 col-md-offset-4">
                <div class="panel panel-default">
                    <div class="panel-heading">
                        <h3 class="panel-title">Please sign in</h3>
                    </div>
                    <div class="panel-body">
<c:url var="loginUrl" value="/login" />
                        <form action="${loginUrl}" method="post"
                            class="form-horizontal">

                            <c:if test="${param.error != null}">
                                <div class="alert alert-danger">
                                    <p>Invalid username and password.
                                    </p>
                                </div>
                            </c:if>

                            <c:if test="${param.logout != null}">
                                <div class="alert alert-success">
                                    <p>You have been logged out
                                    successfully.</p>
                                </div>
                            </c:if>

                            <c:if test="${param.accessDenied !=
```

```
        null}">
        <div class="alert alert-danger">
            <p>Access Denied: You are not
            authorised! </p>
        </div>
    </c:if>

    <div class="input-group input-sm">
        <label class="input-group-addon"
        for="username"><i
            class="fa fa-user"></i></label>
    <input type="text" class="form-control"
        id="userId" name="userId"
        placeholder="Enter Username"
        required>
    </div>
    <div class="input-group input-sm">
        <label class="input-group-addon"
        for="password"><i
            class="fa fa-lock"></i></label>
        <input type="password"
            class="form-control" id="password"
            name="password" placeholder="Enter
            Password" required>
    </div>

    <div class="form-actions">
        <input type="submit"
            class="btn btn-block btn-primary
            btn-default" value="Log in">
    </div>
</form>
</div>
</div>
</div>
</div>
</div>
</body>
```

7. Now create one more configuration file called `SecurityConfig.java` under the `com.packt.webstore.config` package in the `src/main/java` source folder, add the following content into it, and save it:

```
package com.packt.webstore.config;

import org.springframework.beans.factory
    .annotation.Autowired;
import org.springframework.context
```

```
.annotation.Configuration;
import org.springframework.security.config.annotation
.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config
.annotation.web.builders.HttpSecurity;
import org.springframework.security.config
.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config
.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends
WebSecurityConfigurerAdapter {

    @Autowired
    public void
configureGlobalSecurity(AuthenticationManagerBuilder auth)
throws Exception {

auth.inMemoryAuthentication().withUser("john").password("pa55word")
).roles("USER");

auth.inMemoryAuthentication().withUser("admin").password("root123")
).roles("USER", "ADMIN");
    }

    @Override
    protected void configure(HttpSecurity httpSecurity)
throws Exception {

        httpSecurity.formLogin().loginPage("/login")
            .usernameParameter("userId")
            .passwordParameter("password");

        httpSecurity.formLogin().defaultSuccessUrl
("/market/products/add")
            .failureUrl("/login?error");

        httpSecurity.logout().logoutSuccessUrl("/login?
logout");

        httpSecurity.exceptionHandling().accessDeniedPage
("/login?accessDenied");

        httpSecurity.authorizeRequests()
            .antMatchers("/").permitAll()
            .antMatchers("/**/add").access("hasRole('ADMIN')")
    }
```



```
        .antMatchers("/**/market/**").access
        ("hasRole('USER')");

    httpSecurity.csrf().disable();
}
}
```

8. Create one more initializer class called `SecurityWebApplicationInitializer` under the `com.packt.webstore.config` package in the `src/main/java` source folder, add the following content into it, and save it:

```
package com.packt.webstore.config;

import org.springframework.security.web.context
    .AbstractSecurityWebApplicationInitializer;

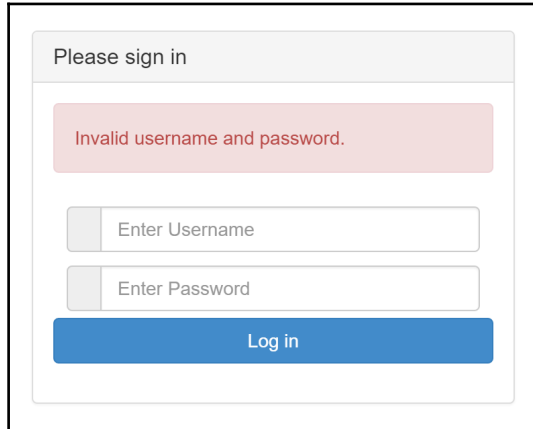
public class SecurityWebApplicationInitializer extends
    AbstractSecurityWebApplicationInitializer {

}
```

9. Now open your `addProduct.jsp` file and add the following code after the `English|Dutch` anchor tag:

```
<a href="<c:url value="/logout" />">Logout</a>
```

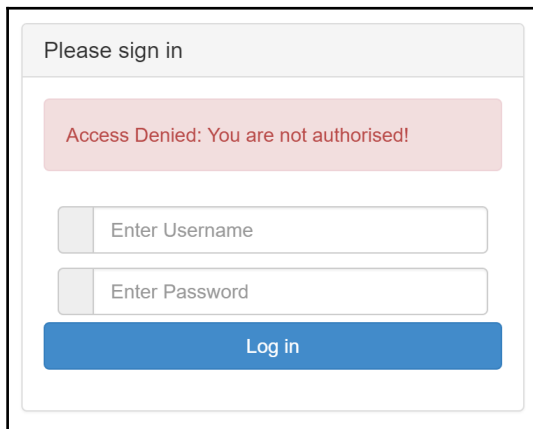
10. Now run your application and enter the URL, `http://localhost:8080/webstore/`; you will see the welcome screen.
11. Now try to access the `products` page by entering the URL, `http://localhost:8080/webstore/market/products`; a login page will be shown. Enter an arbitrary username and password; you will see an **Invalid username and password** error:



A screenshot of a login form titled "Please sign in". The form is enclosed in a light gray border. At the top, there is a red error message box with the text "Invalid username and password.". Below this, there are two input fields: "Enter Username" and "Enter Password", each with a small gray icon to its left. At the bottom of the form is a blue "Log in" button.

Login page showing error messages for invalid credentials

12. Now enter the username `john` and the password `pa55word`, and press the **Log in** button; you should be able to see the regular products page.
13. Now try to access the add products page by entering the URL, `http://localhost:8080/webstore/market/products/add`; again the login page will be shown with the error message **Access Denied: You are not authorised!**



A screenshot of a login form titled "Please sign in". The form is enclosed in a light gray border. At the top, there is a red error message box with the text "Access Denied: You are not authorised!". Below this, there are two input fields: "Enter Username" and "Enter Password", each with a small gray icon to its left. At the bottom of the form is a blue "Log in" button.

Login page showing error messages for unauthorised users

14. Now enter the username `admin` and the password `root123`, and press the **Log in** button; you should be able to see the regular add products page with a **logout** button in the top-right corner.

What just happened?

As usual, in order to use Spring Security in our project, we need some Spring Security-related JARs; from steps 1 to 4 we just added those JARs as Maven dependencies.

In step 5, we created one more controller called `LoginController` to handle all login-related web requests. It simply contains a single request mapping method to handle login, login failure and log out requests. Since the request mapping method returns a view named `login`, we need to create a view file called `login.jsp`, which is what we did in step 6.

`login.jsp` contains many tags with the `bootstrap` style class applied to enhance the look and feel of the login form; we don't need to concentrate on those tags. But some important tags are used to understand the flow; the first one is the `<c:if>` tag:

```
<c:if test="${param.error != null}">
  <div class="alert alert-danger">
    <p>Invalid username and password.</p>
  </div>
</c:if>
```

`<c:if >` is a special JSTL tag to check a condition; it is more like an `if...else` condition that we use in our programming language. Using this `<c:if>` tag we are simply checking whether the page request parameter contains a variable called `error`; if the request parameter contains a variable called `error` we simply show an error message, **Invalid username and password**, within the `<p>` tag using the `<spring:message>` tag.

Similarly, we are also checking whether the request parameter contains variables called `logout` and `accessDenied`; if so we show the corresponding message, also within the `<p>` tag:

```
<c:if test="${param.logout != null}">
  <div class="alert alert-success">
    <p>You have been logged out successfully.</p>
  </div>
</c:if>

<c:if test="${param.accessDenied != null}">
  <div class="alert alert-danger">
    <p>Access Denied: You are not authorised! </p>
```

```
</div>
</c:if>
```

So now we facilitated all possible login-related messages being shown in the login page. The other important tag in `login.jsp` is the `form` tag, which represents the login form. Notice the `action` attribute of the `form` tag:

```
<c:url var="loginUrl" value="/login" />
<form action="${loginUrl}" method="post" class="form-horizontal">
```

We are simply posting our login form values, such as username and password, to the Spring Security authentication handler URL, which is stored in the variable called `${loginUrl}`. Here the special JSTL tag `<c:url>` is used to encode the URL.

Okay, now we have created a controller (`LoginController`) to dispatch the login page. But we need to tell Spring to present this login page to users if they try to access a page without logging in. How can we enforce that? This is where the `WebSecurityConfigurerAdapter` class comes in; by extending `WebSecurityConfigurerAdapter`, we can configure the `HttpSecurity` object for various security-related settings in our web application. So in step 7, we are simply creating a class called `SecurityConfig` to configure the security-related aspects of our web application.

One of the important methods in the `SecurityConfig` class is `configureGlobalSecurity`; under this method we are simply configuring `AuthenticationManagerBuilder` to create two users, `john` and `admin`, with a specified password and roles:

```
@Autowired
public void configureGlobalSecurity(AuthenticationManagerBuilder auth)
throws Exception {
    auth.inMemoryAuthentication().withUser("john")
        .password("pa55word")
        .roles("USER");

    auth.inMemoryAuthentication().withUser("admin")
        .password("root123")
        .roles("USER", "ADMIN");
}
```

The next important method is `configure`; within this method we are doing some authentication-and authorization-related configuration and we will see these one by one. The first configuration tells Spring MVC that it should redirect the users to the login page if authentication is required; here the `loginpage` attribute denotes to which URL it should forward the request to get the login form.

Remember this request path should be the same as the request mapping of the `login()` method of `LoginController`. We are also setting the user name parameter and password parameter name in this configuration:

```
httpSecurity.formLogin().loginPage("/login")
    .usernameParameter("userId")
    .passwordParameter("password");
```

With this configuration, while posting the username and password to the Spring Security authentication handler through the login page, Spring expects those values to be bound under the variable name `userId` and `password` respectively; that's why, if you notice, the input tags for username and password carry the name attributes `userId` and `password` in step 6:

```
<input type="text" class="form-control" id="userId" name="userId"
placeholder="Enter Username" required>
<input type="password" class="form-control" id="password" name="password"
placeholder="Enter Password" required>
```

Similarly, Spring handles the log out operation under the `/logout` URL; that's why in step 9 we formed the logout link on the add products page, as follows:

```
<a href="<c:url value="/logout" />">Logout</a>
```

Next, we are just configuring the default success URL, which denotes the default landing page after a successful login; similarly the authentication failure URL indicates to which URL the request needs to be forwarded in the case of login failure:

```
httpSecurity.formLogin().defaultSuccessUrl("/market/products/add")
    .failureUrl("/login?error");
```

Notice we are setting the request parameter to `error` in the failure URL; thus when the login page is rendered, it will show the error message **Invalid username and password** in the case of login failure. Similarly, we can also configure the logout success URL, which denotes where the request needs to be forwarded to after a logout:

```
httpSecurity.logout().logoutSuccessUrl("/login?logout");
```

You can also see that we are setting the request parameter as `logout` for the logout success URL to match the condition checking that we performed in `login.jsp`:

```
<c:if test="${param.logout != null}">
    <div class="alert alert-success">
        <p>You have been logged out successfully.</p>
    </div>
</c:if>
```

And similarly, we also configured the redirection URL for the access denied page in the case of authorization failure as follows:

```
httpSecurity.exceptionHandling().accessDeniedPage("/login?accessDenied");
```

The request parameter we are setting here for the access denied page URL should match the parameter name that we are checking in the `login.jsp` page:

```
<c:if test="${param.accessDenied != null}">
  <div class="alert alert-danger">
    <p>Access Denied: You are not authorised! </p>
  </div>
</c:if>
```

So now we have configured almost all redirection URLs that specify to which page the user should get redirected in the case of login success, login failure, logout success, and access denial. The next configuration is the more important as it defines which user should get access to which page:

```
httpSecurity.authorizeRequests()
    .antMatchers("/").permitAll()
    .antMatchers("/**/add").access("hasRole('ADMIN')")
    .antMatchers("/**/market/**").access("hasRole('USER')");
```

The preceding configuration defines three important authorization rules for our web application, in terms of Ant pattern matchers. The first one allows the request URLs that end with `/`, even if the request doesn't carry any roles. The next rule allows all request URLs that end with `/add`, if the request has the role `ADMIN`. The third rule allows all request URLs that have the path `/market/` if they have the role `USER`.

Okay we defined all security-related configurations in the security context file, but Spring should know about this configuration file and will have to read this configuration file before booting the application. Only then can it create and manage those security-related configurations. How can we instruct Spring to pick up this file? The answer is `AbstractSecurityWebApplicationInitializer`; by extending the `AbstractSecurityWebApplicationInitializer` class, we can instruct Spring MVC to pick up our `SecurityConfig` class during bootup. So in step 8, that's what we are doing.

After finishing all the steps, if you run your application and enter the `http://localhost:8080/webstore/` URL, you are able to see the welcome screen. Now try to access the `products` page by entering the `http://localhost:8080/webstore/market/products` URL; a login page will be shown. Enter an arbitrary username and password; you will see an **Invalid username and password** error.

Now enter the username `john` and the password `pa55word`, and press the **Log in** button; you should be able to see the regular `products` page. Now try to access the add products page by entering the `http://localhost:8080/webstore/market/products/add` URL; again a login page will be shown with an error message, **Access Denied: You are not authorised!** Now enter the username `admin` and the password `root123`, and press the **Log in** button; you should be able to see the regular add products page with a **logout** button in the top-right corner.

Pop quiz – Spring Security

Which URL is the Spring Security default authentication handler listening on for the username and password?

1. `/login`
2. `/login?userName&password`
3. `/handler/login`

What is the default logout handler URL for Spring Security?

1. `/logout`
2. `/login?logout`
3. `/login?logout=true`

Have a go hero – play with Spring Security

Just for demonstration purposes, I added the **logout** link only on the add products page; why don't you add the **logout** link to every page? Also, try adding new users and assigning new roles to them.

Summary

Web application security is a complex multidimensional problem that needs to be addressed from multiple perspectives such as coding, designing, operations, systems, networking, and policy. What we have seen in this chapter is just a glimpse into Spring Security and how it can be used for authentication and authorization. In the next chapter we will further explore validators.

8

Validate Your Products with a Validator

One of the most commonly expected behaviors of any web application is that it should validate user data. Every time a user submits data into our web application, it needs to be validated. This is to prevent security attacks, wrong data, or simple user errors. We don't have control over what the user may type while submitting data into our web application. For example, they may type some text instead of a date, they may forget to fill a mandatory field, or suppose we used 12-character lengths for a field in the database and the user entered 15-character length data, then the data cannot be saved in the database. Similarly, there are lots of ways a user can feed incorrect data into our web application. If we accept those values as valid, then it will create errors and bugs when we process such inputs. This chapter will explain the basics of setting up validation with Spring MVC.

After finishing this chapter, you will have a clear idea about the following:

- JSR-303 Bean Validation
- Custom validation
- Spring validation

Bean Validation

Java Bean Validation (JSR-303) is a Java specification that allows us to express validation constraints on objects via annotations. It allows the APIs to validate and report violations. **Hibernate Validator** is the reference implementation of the Bean Validation specification. We are going to use Hibernate Validator for validation.

You can see the available Bean Validation annotations at the following site:

<https://docs.oracle.com/javaee/7/tutorial/bean-validation001.htm>.

Time for action – adding Bean Validation support

In this section, we will see how to validate a form submission in a Spring MVC application. In our project, we have the **Add new product** form already. Now let's add some validation to that form:

1. Open `pom.xml`, which you will find under the root directory of the actual project.
2. You will be able to see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as `org.hibernate`, **Artifact Id** as `hibernate-validator`, **Version** as `5.2.4.Final`, select **Scope** as **compile**, click the **OK** button, and save `pom.xml`.
4. Open our `Product` domain class and add the `@Pattern` (`javax.validation.constraints.Pattern`) annotation at the top of the `productId` field, as follows:

```
@Pattern(regexp="P[1-9]+", message="
{Pattern.Product.productId.validation}")
private String productId;
```

5. Similarly, add the `@Size`, `@Min`, `@Digits`, and `@NotNull` (`javax.validation.constraints.*`) annotations on the top of the `name` and `unitPrice` fields respectively, as follows:

```
@Size(min=4, max=50, message="
{Size.Product.name.validation}")
private String name;
@Min(value=0, message="
{Min.Product.unitPrice.validation}")@Digits(integer=8,
fraction=2, message="
{Digits.Product.unitPrice.validation}")@NotNull(message="
{NotNull.Product.unitPrice.validation}")
private BigDecimal unitPrice;
```

6. Open our message source file `messages.properties` from `/src/main/resources` in your project and add the following entries into it:

```
Pattern.Product.productId.validation = Invalid product ID.
It should start with character P followed by number.
```

```
Size.Product.name.validation = Invalid product name. It  
should be minimum 4 characters to maximum 50 characters long.
```

```
Min.Product.unitPrice.validation = Unit price is Invalid. It  
cannot have negative values.
```

```
Digits.Product.unitPrice.validation = Unit price is  
Invalid.It can have maximum of 2 digit fraction and 8 digit  
integer.
```

```
NotNull.Product.unitPrice.validation = Unit price is Invalid.  
It cannot be empty.
```

```
Min.Product.unitPrice.validation = Unit price is Invalid. It  
cannot be negative value.
```

7. Open our `ProductController` class and change the `processAddNewProductForm` request mapping method by adding a `@Valid` (`javax.validation.Valid`) annotation in front of the `newProduct` parameter. After finishing that, your `processAddNewProductForm` method signature should look as follows:

```
public String  
processAddNewProductForm(@ModelAttribute("newProduct")  
@Valid Product newProduct, BindingResult result,  
HttpServletRequest request)
```

8. Now, within the body of the `processAddNewProductForm` method, add the following condition as the first statement:

```
if(result.hasErrors()) {  
    return "addProduct";  
}
```

9. Open our `addProduct.jsp` from `src/main/webapp/WEB-INF/views/` in your project and add the `<form:errors>` tag for the `productId`, `name`, and `unitPrice` input elements. For example, the product ID input tag will have the `<form:errors>` tag beside it, as follows:

```
<form:input id="productId" path="productId" type="text"  
class="form-input-large"/>  
<form:errors path="productId" cssClass="text-danger"/>
```

Remember, the `path` attribute value should be the same as the corresponding input tag.

10. Now, add one global `<form:errors>` tag within the `<form:form>` tag, as follows:

```
<form:errors path="*" cssClass="alert alert-danger"
element="div"/>
```

11. Add bean configuration for `LocalValidatorFactoryBean` in our web application context configuration file `WebApplicationContextConfig.java`, as follows:

```
@Bean(name = "validator")
public LocalValidatorFactoryBean validator() {
    LocalValidatorFactoryBean bean = new
        LocalValidatorFactoryBean();
    bean.setValidationMessageSource(messageSource());
    return bean;
}
```

12. Finally, override the `getValidator` method in `WebApplicationContextConfig` to configure our validator bean as the default validator, as follows:

```
@Override
public Validator getValidator(){
    return validator();
}
```



Note that here the return type
is `org.springframework.validation.Validator`

13. Now run our application and enter the URL `http://localhost:8080/webstore/market/products/add`. We will see a webpage showing a web form to add product info. Without entering any values in the form, simply click on the **Add** button. You will see validation messages at the top of the form, as shown in the following screenshot:

Add new product

Invalid product name. It should be minimum 4 characters to maximum 50 characters long.

Invalid product ID. It should start with character P followed by number.

Unit price is Invalid. It cannot be empty.

New Product ID

Invalid product ID. It should start with character P followed by number.

Name

Invalid product name. It should be minimum 4 characters to maximum 50 characters long.

Unit Price

Unit price is Invalid. It cannot be empty.

Manufacturer

Category

Units in stock

Description

Condition

☐ New ☐ Old ☐ Refurbished

Product Image file

Choose File

No file chosen

Add

The Add new product web form showing validation message.

What just happened?

Since we decided to use the Bean Validation (JSR-303) specification, we needed an implementation of the Bean Validation specification, and we decided to use a Hibernate Validator implementation in our project; thus we need to add that JAR to our project as a dependency. That's what we did in steps 1 through 3.

From steps 4 and 5, we added some `javax.validation.constraints` annotations such as `@Pattern`, `@Size`, `@Min`, `@Digits`, and `@NotNull` on our domain class (`Product.java`) fields. Using these annotations, we can define validation constraints on fields. There are more validation constraint annotations available under the `javax.validation.constraints` package. Just for demonstration purposes, we have used a couple of annotations; you can check out the Bean Validation documentation for all available lists of constraints.

For example, take the `@Pattern` annotation on top of the `productId` field; it will check whether the given value for the field matches the regular expression that is specified in the `regexp` attribute of the `@Pattern` annotation. In our example, we just enforce that the value given for the `productId` field should start with the character `P` and should be followed by digits:

```
@Pattern(regexp="P[1-9]+", message="
{Pattern.Product.productId.validation}")
private String productId;
```

The `message` attribute of every validation annotation is just acting as a key to the actual message from the message source file (`messages.properties`). In our case, we specified `Pattern.Product.productId.validation` as the key, so we need to define the actual validation message in the message source file. That's why we added some message entries in step 6. If you noticed the corresponding value for the key `Pattern.Product.productId.validation` in the `messages.properties` file, you will notice the following value:

```
Pattern.Product.productId.validation = Invalid product ID. It should start
with character P followed by number.
```



You can even add localized error messages in the corresponding message source file if you want. For example, if you want to show error messages in Dutch, simply add error message entries in the `messages_nl.properties` file as well. During validation, this message source will be automatically picked up by Spring MVC based on the chosen locale.

We defined the validation constraints in our domain object, and also defined the validation error messages in our message source file. What else do we need to do? We need to tell our controller to validate the form submission request. We did that through steps 7 and 8 in the `processAddNewProductForm` method:

```
@RequestMapping(value = "/products/add", method = RequestMethod.POST)
public String processAddNewProductForm(@ModelAttribute("newProduct") @Valid
Product newProduct, BindingResult result, HttpServletRequest request) {
    if(result.hasErrors()) {
        return "addProduct";
    }

    String[] suppressedFields = result.getSuppressedFields();
    if (suppressedFields.length > 0) {
        throw new RuntimeException("Attempting to bind disallowed fields: " +
StringUtils.arrayToCommaDelimitedString(suppressedFields));
    }
}
```

```
MultipartFile productImage = newProduct.getProductImage();
String rootDirectory =
request.getSession().getServletContext().getRealPath("/");

    if (productImage!=null && !productImage.isEmpty()) {
        try {
            productImage.transferTo(new
File(rootDirectory+"resources\\images"+ newProduct.getProductid() +
".png"));
        } catch (Exception e) {
            throw new RuntimeException("Product Image saving failed", e);
        }
    }

    productService.addProduct(newProduct);
    return "redirect:/market/products";
}
```

We first annotated our method parameter `newProduct` with the `@Valid` (`javax.validation.Valid`) annotation. By doing so, Spring MVC will use the Bean Validation framework to validate the `newProduct` object. As you already know, the `newProduct` object is our form-backed bean. After validating the incoming form bean (`newProduct`), Spring MVC will store the results in the `result` object; this is again another method parameter of our `processAddNewProductForm` method.

In step 8 we simply checked whether the `result` object contains any errors. If so, we redirected to the same **Add new product** page; otherwise we proceeded to add `productToBeAdded` to our repository.

So far everything is fine. At first we defined the constraints on our domain object and defined the error messages in the message source file (`messages.properties`). Later we validated and checked the validation result in the controller's form processing method (`processAddNewProductForm`), but we haven't said how to show the error messages in the view file. Here comes Spring MVC's special `<form:errors>` tag to the rescue.

We added this tag for the `productId`, `name`, and `unitPrice` input elements in step 9. If any of the input fields failed during validation, the corresponding error message will be picked up by this `<form:errors>` tag:

```
<form:errors path="productId" cssClass="text-danger"/>
```

The `path` attribute is used to identify the field in the form bean to look for errors, and the `cssClass` attribute will be used to style the error message. I have used Bootstrap's style class `text-danger`, but you can use any valid CSS- style class that you prefer to apply on the error message.

Similarly, in step 10, we have added a global `<form:errors>` tag to show all error messages as a consolidated view at the top of the form:

```
<form:errors path="*" cssClass="alert alert-danger" element="div"/>
```

Here we have used the `"*"` symbol for the `path` attribute, which that means we want to show all the errors and element attributes, just indicating which type of element Spring MVC should use to list all the errors.

So far, we have done all the coding-related stuff that is needed to enable validation, but we have to do one final configuration in our web application context to enable validation; that is we need to introduce the Bean Validation framework to our Spring MVC. In steps 11 and 12 we did just that. We have defined a bean for `LocalValidatorFactoryBean` (`org.springframework.validation.beanvalidation.LocalValidatorFactoryBean`). This `LocalValidatorFactoryBean` will initiate the Hibernate Validator during the booting of our application:

```
@Bean(name = "validator")
public LocalValidatorFactoryBean validator() {
    LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
    bean.setValidationMessageSource(messageSource());
    return bean;
}
```

The `setValidationMessageSource` method of `LocalValidatorFactoryBean` indicates which message source bean it should look to for error messages. Since, in Chapter 6, *Internalize Your Store with Interceptor*, we already configured message sources in our web application context, we just use that bean, which is why we assigned the value `messageSource()` as the value for the `setValidationMessageSource` method. You will see a bean definition under the method `messageSource()` already in our web application context.

And finally, we introduced our validator bean to Spring MVC by overriding the `getValidator` method:

```
@Override
public Validator getValidator(){
    return validator();
}
```

That is all we did to enable validation; now, if you run our application and bring the **Add new product** page using the

`http://localhost:8080/webstore/market/products/add` URL, you can see the empty form ready to submit. If you submit that form without filling in any information, you

will see error messages in red.

Have a go hero – adding more validation in the Add new product page

I have just added validation for the first three fields in the `Product` domain class; you can extend the validation for the remaining fields. And try to add localisedlocalized error messages for the validation you are defining.

Here are some hints you can try out:

- Add validation to show a validation message in case the `category` field is empty
- Try to add validation to the `unitsInStock` field to validate that the minimum number of **Units in stock** allowed is zero

Custom validation with JSR-303/Bean Validation

In the previous sections, we learned how to use standard JSR-303 Bean Validation annotations to validate fields of our domain object. This works great for simple validations, but sometimes we need to validate some custom rules that aren't available in the standard annotations. For example, what if we need to validate a newly added product's ID, which should not be the same as any existing product ID? To accomplish this type of thing, we can use custom validation annotations.

Time for action – adding Bean Validation support

In this exercise we are going to learn how to create custom validation annotations and how to use them. Let's add a custom product ID validation to our **Add new product** page to validate duplicate product IDs:.

1. Create an annotation interface called `ProductId` (`ProductId.java`) under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code snippet:

```
package com.packt.webstore.validator;

import static java.lang.annotation.ElementType
```



```
.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = ProductIdValidator.class)
@Documented
public @interface ProductId {
    String message() default "
{com.packt.webstore.validator.ProductId.message}";

    Class<?>[] groups() default {};
    public abstract Class<? extends Payload>[] payload()
default {};
}
```

2. Now, create a class called `ProductIdValidator` under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.validator;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import org.springframework.beans.factory.annotation
.Autowired;
import com.packt.webstore.domain.Product;
import com.packt.webstore.exception.ProductNotFoundException;
import com.packt.webstore.service.ProductService;

public class ProductIdValidator implements
ConstraintValidator<ProductId, String>{
    @Autowired
    private ProductService productService;

    public void initialize(ProductId constraintAnnotation) {
        // intentionally left blank; this is the place to
        initialize the constraint annotation for any sensible default
        values.
    }
}
```

```
    public boolean isValid(String value,
        ConstraintValidatorContext context) {
        Product product;
        try {
            product = productService.getProductById(value);
        } catch (ProductNotFoundException e) {
            return true;
        }
        if(product != null) {
            return false;
        }
        return true;
    }
}
```

3. Open our message source file `messages.properties` from `/src/main/resources` in your project and add the following entry into it:

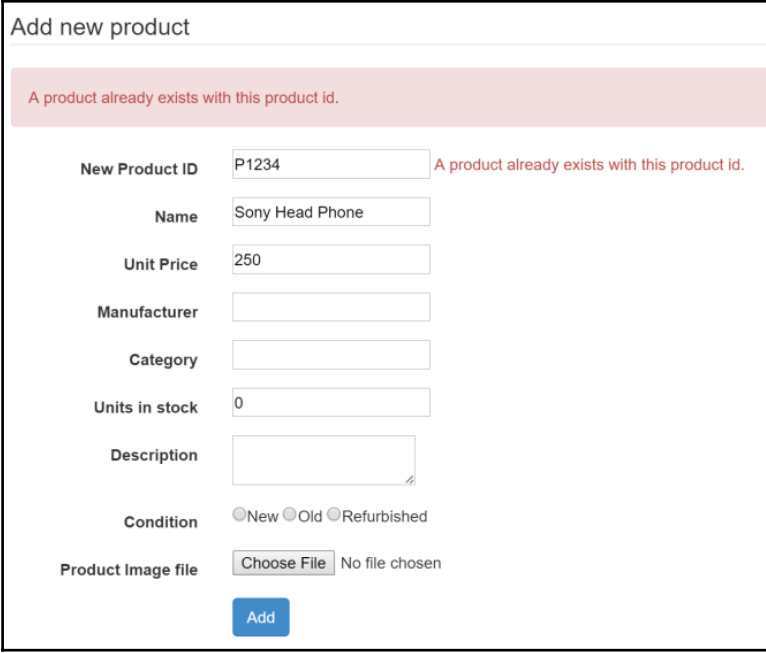
```
com.packt.webstore.validator.ProductId.message = A product
already exists with this product id.
```

4. Finally, open our `Product` (`Product.java`) domain class and annotate our `productId` field with our newly created `ProductId` annotation, as follows:

```
@Pattern(regexp="P[1-9]+", message="
{Pattern.Product.productId.validation}")
@ProductId
private String productId;
```

5. Now run our application and enter the URL `http://localhost:8080/webstore/market/products/add`. You should be able to see a webpage showing a web form to add product info. Enter all the values in the form, particularly filling in the product ID field with the value `P1234`, and simply click the **Add** button.

6. You will see validation messages at the top of the form, as shown in the following screenshot:



The screenshot shows a web form titled "Add new product". At the top, there is a red error message: "A product already exists with this product id." Below this, the form contains several input fields: "New Product ID" (with value "P1234" and a red error message "A product already exists with this product id."), "Name" (with value "Sony Head Phone"), "Unit Price" (with value "250"), "Manufacturer", "Category", "Units in stock" (with value "0"), and "Description". There are also radio buttons for "Condition" (New, Old, Refurbished) and a "Product Image file" section with a "Choose File" button and the text "No file chosen". An "Add" button is at the bottom.

The Add new product web form showing custom validation.

What just happened?

In step 1 we just created our custom validation annotation called `ProductId`. Every custom validation annotation we create needs to be annotated with the `@Constraint` (`javax.validation.Constraint`) annotation. The `@Constraint` annotation has an important property called `validatedBy`, which indicates the class that is performing the actual validation. In our case, we have given a value `ProductIdValidator.class` for the `validatedBy` property. So our `ProductId` validation annotation would expect a class called `ProductIdValidator`. That's why in step 2 we have created the class `ProductIdValidator` by implementing the interface `ConstraintValidator` (`javax.validation.ConstraintValidator`).

We annotated the `ProductIdValidator` class with the `@Component` (`org.springframework.stereotype.Component`) annotation; the `@Component` annotation is another stereotype annotation that is available in Spring. It is very similar to the `@Repository` or `@Service` annotations; during the booting of our application, Spring would create and maintain an object for the `ProductIdValidator` class. So `ProductIdValidator` will become a managed bean in our web application context, which is the reason we were able to autowire the `productService` bean in `ProductIdValidator`.

Next we autowired the `ProductService` object in the `ProductIdValidator` class. Why? Because, inside the `isValid` method of the `ProductIdValidator` class, we have used the `productService` to check whether any product with the given ID exists:

```
public boolean isValid(String value, ConstraintValidatorContext context) {
    Product product;
    try {
        product = productService.getProductById(value);
    } catch (ProductNotFoundException e) {
        return true;
    }
    if(product != null) {
        return false;
    }
    return true;
}
```

If any product exists with the given product ID, we are invalidating the validation by returning false, otherwise we are passing the validation by returning true.

In step 3 we just added our default error message for our custom validation annotation in the message source file (`messages.properties`). If you observed carefully, the key (`com.packt.webstore.validator.ProductId.message`) we have used in our message source file is the same as the default key that we have defined in the `ProductId` (`ProductId.java`) validation annotation:

```
String message() default
"{com.pactk.webstore.validator.ProductId.message}";
```

So, finally, in step 4 we have used our newly created `ProductId` validation annotation in our domain class (`Product.java`). So it will act similarly to any other JSR-303 Bean Validation annotation.

Thus, you were able to see the error message on the screen when you entered the existing product ID as the product ID for the newly added product.

Have a go hero – adding custom validation to a category

Create a custom validation annotation called `@Category`, which will allow only some of the predefined configured categories to be entered. Consider the following things while implementing your custom annotation:

- Create an annotation interface called `CategoryValidator` under the package `com.packt.webstore.validator`
- Create a corresponding `ConstraintValidator` called `CategoryValidator` under the package `com.packt.webstore.validator`
- Add a corresponding error message in the message source file
- Your `CategoryValidator` should maintain a list of allowed categories (`List<String> allowedCategories`) to check whether the given category exists under the list of allowed categories
- Don't forget to initialize the `allowedCategories` list in the constructor of the `CategoryValidator` class
- Annotate the `category` field of the `Product` domain class with the `@Category` annotation

After applying your custom validation annotation `@category` on the `category` field of the `Product` domain class, your **Add new product** page should reject products of categories that have been not configured in the `CategoryValidator`.

Spring validation

We have seen how to incorporate JSR-303 Bean Validation with Spring MVC. In addition to Bean Validation, Spring has its own classic mechanism to perform validation as well, which is called Spring `vValidation`. The JSR-303 Bean Validation is much more elegant, expressive, and, in general, simpler to use when compared to the classic Spring validation. But the classic Spring validation is very flexible and extensible. For example, consider a cross field validation where we want to compare two or more fields to see whether their values can be considered as valid in combination. In such a case we can use Spring validation.

In the last section, using JSR-303 Bean Validation we have validated some of the individual fields on our `Product` domain object; we haven't done any validation that combines one or more fields. We don't know whether the combination of different fields makes sense.

Time for action – adding Spring validation

For example, say we have a constraint that we should not allow more than 99 units of any product to be added if the unit price is greater than \$ 1,000 for that product. Let's see how to add such a validation using Spring validation in our project:

1. Create a class called `UnitsInStockValidator` under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.validator;

import java.math.BigDecimal;
import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;
import com.packt.webstore.domain.Product;

@Component
public class UnitsInStockValidator implements Validator{

    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        Product product = (Product) target;

        if(product.getUnitPrice() != null && new
        BigDecimal(1000).compareTo(product.getUnitPrice()) <= 0 &&
        product.getUnitsInStock() > 99) {
            errors.rejectValue("unitsInStock",
            "com.packt.webstore.validator
            .UnitsInStockValidator.message");
        }
    }
}
```

2. Open our message source file `messages.properties` from `/src/main/resources` in your project and add the following entry into it:

```
com.packt.webstore.validator.UnitsInStockValidator.message =
You cannot add more than 99 units if the unit price is
greater than 1000.
```

3. Open the `ProductController` class and autowire a reference to the `UnitsInStockValidator` class as follows:

```
@Autowired
private UnitsInStockValidator unitsInStockValidator;
```

4. Now, inside the `initialiseBinder` method in the `ProductController` class, add the following line:

```
binder.setValidator(unitsInStockValidator);
```

5. Now run our application and enter `http://localhost:8080/webstore/market/products/add`, and you will be able to see a webpage showing a web form to add product info. Enter all the valid values in the form; particularly fill in the **Unit Price** field with the value 1500 and **Units in stock** field with the value 150. Now simply click on the **Add** button and you will see validation messages at the top of the form as shown in the following screenshot:

Add new product

You cannot add more than 99 units if the unit price is greater than 1000.

New Product ID

Name

Unit Price

Manufacturer

Category

Units in stock

Description

Condition ☒ New ☐ Old ☐ Refurbished

Product Image file No file chosen

The Add new product web form showing cross field validation.

What just happened?

In classic Spring validation, the main validation construct is the `Validator` (`org.springframework.validation.Validator`) interface. The `Spring Validator` interface defines two methods for validation purposes, namely `supports` and `validate`. The `supports` method indicates whether the validator can validate a specific class. If so, the `validate` method can be called to validate an object of that class.

Every Spring- based validator we are creating should implement this interface. In step 1 we just did just that; we simply created a class called `UnitsInStockValidator`, which implements the `Spring Validator` interface.

Inside the `validate` method of the `UnitsInStockValidator` class, we simply check whether the given `Product` object has a unit price greater than 1,000 and the number of units in stock is more than 99; if so, we reject that value with the corresponding error key to show the error message from the message source file:

```
@Override
public void validate(Object target, Errors errors) {
    Product product = (Product) target;

    if(product.getUnitPrice() != null && new
    BigDecimal(1000).compareTo(product.getUnitPrice())<=0 &&
    product.getUnitsInStock()>99) {
        errors.rejectValue("unitsInStock",
        "com.packt.webstore.validator.UnitsInStockValidator.message");
    }
}
```

In step 2 we simply added the actual error message for the error key `com.packt.webstore.validator.UnitsInStockValidator.message` in the message source file (`messages.properties`).

We created the validator but, to kick in the validation, we need to associate that validator with the controller. That's what we did in steps 3 and 4. In step 3 we simply added and autowired the reference to `UnitsInStockValidator` in the `ProductController` class. And we associated the `unitsInStockValidator` with `WebDataBinder` in the `initialiseBinder` method:

```
@InitBinder
public void initialiseBinder(WebDataBinder binder) {

    binder.setValidator(unitsInStockValidator);

    binder.setAllowedFields("productId",
```



```
        "name",  
        "unitPrice",  
        "description",  
        "manufacturer",  
        "category",  
        "unitsInStock",  
        "condition",  
        "productImage",  
        "language");  
    }
```

That's it, we created and configured our Spring-based validator to do the validation. Now run our application and enter

`http://localhost:8080/webstore/market/products/add` to show the web form for adding product info. Fill all the values in the form, particularly fill the **Unit Price** field with the value 1000 and **Units in stock** field with the value 150, and click on the **Add** button. You will see validation messages at the top of the form saying **You cannot add more than 99 units if the unit price is greater than 1000**.

It is good that we have added Spring-based validation into our application. But since we have configured our Spring-based validator (`unitsInStockValidator`) with `WebDataBinder`, the Bean Validation that we have configured earlier would not take effect. Spring MVC will simply ignore those JSR-303 Bean Validation annotations (`@Pattern`, `@Size`, `@Min`, `@Digits`, `@NotNull`, and more).

Time for action – combining Spring validation and Bean Validation

So we need to write those Bean Validations again in classic Spring-based validation. This is not a good idea but, thanks to the flexibility and extensibility of Spring validation, we can combine both Spring-based validation and Bean Validation together with a little extra code. Let's do that:

1. Create a class called `ProductValidator` under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.validator;  
  
import java.util.HashSet;  
import java.util.Set;  
import javax.validation.ConstraintViolation;  
import org.springframework.beans.factory
```

```
.annotation.Autowired;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;
import com.packt.webstore.domain.Product;

public class ProductValidator implements Validator{

    @Autowired private javax.validation
    .Validator beanValidator;

    private Set<Validator> springValidators;

    public ProductValidator() {
        springValidators = new HashSet<Validator>();
    }

    public void setSpringValidators(Set<Validator>
springValidators) {
        this.springValidators = springValidators;
    }

    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        Set<ConstraintViolation<Object>> constraintViolations =
beanValidator.validate(target);

        for (ConstraintViolation<Object> constraintViolation :
constraintViolations) {
            String propertyPath =
constraintViolation.getPropertyPath().toString();
            String message = constraintViolation.getMessage();
            errors.rejectValue(propertyPath, "", message);
        }

        for(Validator validator: springValidators) {
            validator.validate(target, errors);
        }
    }
}
```

2. Now open our web application context configuration file `WebApplicationContextConfig.java` and add the following bean definition into it:

```
@Bean
public ProductValidator productValidator () {
    Set<Validator> springValidators = new HashSet<>();
    springValidators.add(new UnitsInStockValidator());
    ProductValidator productValidator = new
    ProductValidator();
    productValidator.setSpringValidators(springValidators);
    return productValidator;
}
```

3. Open our `ProductController` class and replace the existing reference of the `UnitsInStockValidator` field with our newly created `ProductValidator` class, as follows:

```
@Autowired
private ProductValidator productValidator;
```

4. Now, inside the `initialiseBinder` method of the `ProductController` class, replace the `binder.setValidator(unitsInStockValidator)` statement with the following statement:

```
binder.setValidator(productValidator);
```

5. Now run our application and enter the URL `http://localhost:8080/webstore/market/products/add` to check whether all the validations are working fine. Just click the **Add** button without filling anything in on the form. You will notice Bean Validation taking place; similarly fill the **Unit Price** field with the value 1500 and the **Units in stock** field with the value 150 to see Spring validation, as shown in the following screenshot:

Add new product

Invalid product name. It should be minimum 4 characters to maximum 50 characters long.
A product already exists with this product id.
You cannot add more than 99 units if the unit price is greater than 1000.

New Product ID

P1234

A product already exists with this product id.

Name

Invalid product name. It should be minimum 4 characters to maximum 50 characters long.

Unit Price

1500

Manufacturer

Sony

Category

Head phones

Units in stock

150

Description

With built-in noise canceling technology

Condition

☒New ☐Old ☐Refurbished

Product Image file

Choose File

No file chosen

Add

The Add new product web form showing Bean Validation and Spring validation together

What just happened?

Well, our aim was to combine Bean Validations and our Spring-based validation (`unitsInStockValidator`) together, to create a common adapter validator called `ProductValidator`. If you notice closely, the `ProductValidator` class is nothing but an implementation of the regular Spring validator.

We have autowired our existing bean validator into the `ProductValidator` class through the following line:

```
@Autowired
private javax.validation.Validator beanValidator;
```

Later, we used this `beanValidator` reference inside the `validate` method of the `ProductValidator` class to validate all Bean Validation annotations, as follows:

```
Set<ConstraintViolation<Object>> constraintViolations =
beanValidator.validate(target);
```

```
for (ConstraintViolation<Object> constraintViolation :
constraintViolations) {
    String propertyPath = constraintViolation.getPropertyPath().toString();
    String message = constraintViolation.getMessage();
    errors.rejectValue(propertyPath, "", message);
}
```

The `beanValidator.validate(target)` statement returned all the constraint violations. Then, using the `errors` object, we threw all the invalid constraints as error messages. So every Bean Validation annotation we specified in the `Product` domain class will get handled within a `for` loop.

Similarly, we have one more `for` loop to handle all Spring validations in the `validate` method of the `ProductValidator` class:

```
for(Validator validator: springValidators) {
    validator.validate(target, errors);
}
```

This `for` loop iterates through the set of Spring validators and validates them one by one, but if you notice, we haven't initiated the `springValidators` reference. Thus, you may wonder where we have initiated the `springValidators` set. You can find the answer in step 2; we have created a bean for the `ProductValidator` class in our web application context (`WebApplicationContextConfig.java`) and assigned the `springValidators` set:

```
@Bean
public ProductValidator productValidator () {
    Set<Validator> springValidators = new HashSet<>();
    springValidators.add(new UnitsInStockValidator());

    ProductValidator productValidator = new ProductValidator();
    productValidator.setSpringValidators(springValidators);

    return productValidator;
}
```

So now we have created a common adapter validator that can adopt Bean Validation and Spring validation, and validates all Spring and Bean-based validations together. Now we have to replace the `UnitsInStockValidator` reference with the `ProductValidator` reference in our `ProductController` class to kick in our new `ProductValidator`, which we have done in steps 3 and 4. We simply replaced the `UnitsInStockValidator` with `ProductValidator` in the binder, as follows:

```
@InitBinder
```

```
public void initialiseBinder(WebDataBinder binder) {

    binder.setValidator(productValidator);

    binder.setAllowedFields("productId",
        "name",
        "unitPrice",
        "description",
        "manufacturer",
        "category",
        "unitsInStock",
        "condition",
        "productImage",
        "language");

}
```

So we have successfully configured our newly created `ProductValidator` with `ProductController`. To see it in action, just run our application and enter the URL `http://localhost:8080/webstore/market/products/add`. Then, enter some invalid values such as an existing product ID, or fill the **Unit Price** field with the value 1000 and the **Units in stock** field with the value 100. You will notice the Bean Validation error messages and Spring validation error messages on the screen.

Have a go hero – adding Spring validation to a product image

Create a Spring validation class called `ProductImageValidator`, which will validate the size of the product image. It should only allow images whose size is less than or equal to the predefined configured size. Consider the following things while implementing your `ProductImageValidator`:

- Create a validation class called `ProductImageValidator` under the package `com.packt.webstore.validator` by implementing the `org.springframework.validation.Validator` interface
- Add a corresponding error message in the message source file
- Your `ProductImageValidator` should maintain a long variable called `allowedSize` to check whether the given image size is less than or equal to it

- Create a bean for the `ProductImageValidator` class in the servlet context and add it under the `springValidators` set of the `productValidtor` bean
- Remember, don't forget to set the `allowedSize` property in the `ProductImageValidator` bean

After applying your custom validation annotation `@category` on the `category` field of the `Product` domain class, your **Add new product** page should reject products of other categories that have been not configured in the `CategoryValidator`.

Summary

In this chapter, you learned the concept of validation and saw how to enable Bean Validation in Spring MVC for form processing. We also learned how to set up a custom validation using the extension capability of the Bean Validation framework. After that, we learned how to do cross field validation using Spring validation. Finally, we saw how to integrate Bean Validation and Spring validation together.

In our next chapter, we will see how to develop an application in RESTful services. We will be covering the basic concepts of HTTP verbs and will try to understand how they are related to standard CRUD operations. We will also cover how to fire an Ajax request and how to handle it.

9

Give REST to Your Application with Ajax

REST stands for REpresentational State Transfer; REST is an architectural style. Everything in REST is considered as a resource and every resource is identified by a URI. RESTful web services have been embraced by large service providers across the Web as an alternative to SOAP-based Web Services due to their simplicity.

After finishing this chapter, you will have a clear idea about:

- REST web services
- Ajax

Introduction to REST

As I already mentioned, in a REST-based application, everything including static resources, data, and operations is considered as a resource and identified by a URI. For example, consider a piece of functionality to add a new product to our store; we can represent that operation by a URI, something such as

`http://localhost:8080/webstore/products/add`, and we can pass the new product details in XML or JSON representation to that URL. So, in REST, URIs are used to connect clients and servers to exchange resources in the form of representations (HTML, XML, JSON, and so on). In order to exchange data, REST relies on basic HTTP protocol methods GET, POST, PUT, and DELETE.

Spring provides extensive support for developing REST-based web services. In our previous chapters, we have seen that, whenever a web request comes in, we returned a web page to serve that request; usually that web page contained some states (that is, dynamic data). However, in REST-based applications, we only return the states and it is up to the client to decide how to render or present the data to the end user.

Usually, REST-based web services return data in two formats, XML and JSON. We are going to develop some REST-based web services that can return data in the JSON format. After we get the JSON data, we are going to render that data as an HTML page using some JavaScript libraries in the browser.

In our `webstore` application, we have successfully listed some of the products, but the store cannot make a profit without enabling the end user to pick up some products in his/her shopping cart. So let's add a shopping cart facility to our store.

Time for action – implementing RESTful web services

We are going to add the shopping cart facility in two stages. Firstly, we will create a REST-style Controller to handle all shopping cart-related web service requests. Secondly, we will add some JavaScript code to render the JSON data returned by the REST web Service Controller. So first, let's implement some RESTful web services using Spring MVC Controllers, so that later we can add some JavaScript code to consume those web services:

1. Add the following schema definitions for `CART` and `CART_ITEM` tables in `create-table.sql`. You can find `create-table.sql` under the folder `structure src/main/resources/db/sql`:

```
CREATE TABLE CART (
    ID VARCHAR(50) PRIMARY KEY
);

CREATE TABLE CART_ITEM (
    ID VARCHAR(75),
    PRODUCT_ID VARCHAR(25) FOREIGN KEY REFERENCES
PRODUCTS(ID),
    CART_ID varchar(50) FOREIGN KEY REFERENCES
CART(ID),
    QUANTITY BIGINT,
    CONSTRAINT CART_ITEM_PK PRIMARY KEY (ID,CART_ID)
);
```

2. Add the following drop table command as the first line in `create-table.sql`:

```
DROP TABLE CART_ITEM IF EXISTS;  
DROP TABLE CART IF EXISTS;
```

3. Create a domain class named `CartItem` under the package `com.packt.webstore.domain` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.domain;  
  
import java.io.Serializable;  
import java.math.BigDecimal;  
  
public class CartItem implements Serializable{  
  
    private static final long serialVersionUID = -  
        4546941350577482213L;  
  
    private String id;  
    private Product product;  
    private int quantity;  
    private BigDecimal totalPrice;  
  
    public CartItem(String id) {  
        super();  
        this.id = id;  
    }  
  
    public String getId() {  
        return id;  
    }  
  
    public Product getProduct() {  
        return product;  
    }  
  
    public void setProduct(Product product) {  
        this.product = product;  
        this.updateTotalPrice();  
    }  
  
    public int getQuantity() {  
        return quantity;  
    }  
  
    public void setQuantity(int quantity) {
```

```
        this.quantity = quantity;
    }

    public BigDecimal getTotalPrice() {
        this.updateTotalPrice();
        return totalPrice;
    }

    public void updateTotalPrice() {
        totalPrice =
this.product.getUnitPrice().multiply(new
BigDecimal(this.quantity));
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 :
id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        CartItem other = (CartItem) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}
```

4. Similarly, add one more domain class named `Cart` in the same package and add the following code into it:

```
package com.packt.webstore.domain;

import java.io.Serializable;
import java.math.BigDecimal;
```

```
import java.util.List;
import java.util.function.Function;

public class Cart implements Serializable{

    private static final long serialVersionUID =
6554623865768217431L;

    private String id;
    private List<CartItem> cartItems;
    private BigDecimal grandTotal;

    public Cart(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public BigDecimal getGrandTotal() {
        updateGrandTotal();
        return grandTotal;
    }

    public void setGrandTotal(BigDecimal grandTotal) {
        this.grandTotal = grandTotal;
    }

    public List<CartItem> getCartItems() {
        return cartItems;
    }

    public void setCartItems(List<CartItem> cartItems) {
        this.cartItems = cartItems;
    }

    public CartItem getItemByProductId(String
productId) {
        return cartItems.stream().filter(cartItem ->
cartItem.getProduct().getProductId()
.equals(productId))
                        .findAny()
                        .orElse(null);
    }

    public void updateGrandTotal() {
```

```
        Function<CartItem, BigDecimal> totalMapper =
        cartItem -> cartItem.getTotalPrice();

        BigDecimal grandTotal = cartItems.stream()
            .map(totalMapper)
            .reduce(BigDecimal.ZERO, BigDecimal::add);

        this.setGrandTotal(grandTotal);
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 :
id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Cart other = (Cart) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}
```

5. Create a data transfer object (**dto**) named `CartItemDto` under the package `com.packt.webstore.dto` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.dto;

import java.io.Serializable;

public class CartItemDto implements Serializable{

    private static final long serialVersionUID = -
```

```
3551573319376880896L;

    private String id;
    private String productId;
    private int quantity;
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

6. Similarly, add one more dto object named `CartDto` in the same package and add the following code into it:

```
package com.packt.webstore.dto;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class CartDto implements Serializable{

    private static final long serialVersionUID = -
2017182726290898588L;

    private String id;
    private List<CartItemDto> cartItems;

    public CartDto() {}
```

```
public CartDto(String id) {
    this.id = id;
    cartItems = new ArrayList<>();
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public List<CartItemDto> getCartItems() {
    return cartItems;
}

public void setCartItems(List<CartItemDto>
cartItems) {
    this.cartItems = cartItems;
}

public void addCartItem(CartItemDto cartItemDto) {
    this.cartItems.add(cartItemDto);
}
}
```

7. Create a class named `CartItemMapper` under the package `com.packt.webstore.domain.repository.impl` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.domain.repository.impl;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import com.packt.webstore.domain.CartItem;
import com.packt.webstore.service.ProductService;

public class CartItemMapper implements
RowMapper<CartItem> {
    private ProductService productService;
    public CartItemMapper(ProductService
productService) {
        this.productService = productService;
    }
}
```

```
    }

    @Override
    public CartItem mapRow(ResultSet rs, int rowNum)
    throws SQLException {
        CartItem cartItem = new
        CartItem(rs.getString("ID"));
        cartItem.setProduct(productService.getProductById
        (rs.getString("PRODUCT_ID")));
        cartItem.setQuantity(rs.getInt("QUANTITY"));

        return cartItem;
    }
}
```

8. Create a class named `CartMapper` under the package `com.packt.webstore.domain.repository.impl` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.domain.repository.impl;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam
.NamedParameterJdbcTemplate;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.domain.CartItem;
import com.packt.webstore.service.ProductService;

public class CartMapper implements RowMapper<Cart> {
    private CartItemMapper cartItemMapper;
    private NamedParameterJdbcTemplate jdbcTemplate;
    public CartMapper(NamedParameterJdbcTemplate
    jdbcTemplate, ProductService productService) {
        this.jdbcTemplate = jdbcTemplate;
        cartItemMapper = new
        CartItemMapper(productService);
    }

    public Cart mapRow(ResultSet rs, int rowNum)
    throws SQLException {
        String id = rs.getString("ID");
        Cart cart = new Cart(id);
```



```
        String SQL = String.format("SELECT * FROM  
CART_ITEM WHERE CART_ID = '%s'", id);  
        List<CartItem> cartItems =  
jdbcTemplate.query(SQL, cartItemMapper);  
        cart.setCartItems(cartItems);  
        return cart;  
    }  
}
```

9. **Create an interface named `CartRepository` under the package `com.packt.webstore.domain.repository` in the source folder `src/main/java` and add the following method declarations into it:**

```
package com.packt.webstore.domain.repository;  
  
import com.packt.webstore.domain.Cart;  
import com.packt.webstore.dto.CartDto;  
  
public interface CartRepository {  
  
    void create(CartDto cartDto);  
    Cart read(String id);  
    void update(String id, CartDto cartDto);  
    void delete(String id);  
  
    void addItem(String cartId, String productId);  
  
    void removeItem(String cartId, String productId);  
}
```

10. **Create an implementation class named `InMemoryCartRepository` for the previous interface under the package `com.packt.webstore.domain.repository.impl` in the source folder `src/main/java` and add the following code into it:**

```
package com.packt.webstore.domain.repository.impl;  
  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
import org.springframework.beans.factory  
.annotation.Autowired;  
import org.springframework.dao  
.EmptyResultDataAccessException;  
import org.springframework.jdbc.core.namedparam  
.NamedParameterJdbcTemplate;
```

```
import org.springframework.stereotype.Repository;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.domain.CartItem;
import com.packt.webstore.domain.Product;
import com.packt.webstore.domain.
repository.CartRepository;
import com.packt.webstore.
dto.CartDto;
import com.packt.webstore.dto.CartItemDto;
import com.packt.webstore.service.ProductService;

@Repository
public class InMemoryCartRepository implements
CartRepository{
    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;
    @Autowired
    private ProductService productService;
    public void create(CartDto cartDto) {
        String INSERT_CART_SQL = "INSERT INTO CART(ID)
VALUES (:id)";

        Map<String, Object> cartParams = new
HashMap<String, Object>();
cartParams.put("id", cartDto.getId());
        jdbcTemplate.update(INSERT_CART_SQL,
cartParams);

cartDto.getCartItems().stream()
    .forEach(cartItemDto ->{
        Product productById =
productService.getProductById
(cartItemDto.getProductId());
        String INSERT_CART_ITEM_SQL =
"INSERT INTO CART_ITEM(ID,PRODUCT_ID
,CART_ID,QUANTITY) "
                                + "VALUES (:id,
:product_id, :cart_id, :quantity)";

        Map<String, Object> cartItemsParams = new
HashMap<String, Object>();
        cartItemsParams.put("id",
cartItemDto.getId());
        cartItemsParams.put("product_id",
productById.getProductId());
        cartItemsParams.put("cart_id",
cartDto.getId());
```

```
        cartItemsParams.put("quantity",
        cartItemDto.getQuantity());
        jdbcTemplate.update(INSERT_CART_ITEM_SQL,
        cartItemsParams);
    });
}
public Cart read(String id) {
    String SQL = "SELECT * FROM CART WHERE ID = :id";
    Map<String, Object> params = new HashMap<String,
Object>();
    params.put("id", id);
    CartMapper cartMapper = new
    CartMapper(jdbcTemplate, productService);
    try {
        return jdbcTemplate.queryForObject(SQL,
        params, cartMapper);
    } catch (EmptyResultDataAccessException e) {
        return null;
    }
}
@Override
public void update(String id, CartDto cartDto) {
    List<CartItemDto> cartItems =
    cartDto.getCartItems();
    for(CartItemDto cartItem :cartItems) {
        String SQL = "UPDATE CART_ITEM SET QUANTITY
= :quantity, PRODUCT_ID = :productId WHERE ID = :id
AND CART_ID = :cartId";
        Map<String, Object> params = new
        HashMap<String, Object>();
        params.put("id", cartItem.getId());
        params.put("quantity",
        cartItem.getQuantity());
        params.put("productId",
        cartItem.getProductId());
        params.put("cartId", id);
        jdbcTemplate.update(SQL, params);
    }
}

@Override
public void delete(String id) {
    String SQL_DELETE_CART_ITEM = "DELETE FROM
CART_ITEM WHERE CART_ID = :id";
    String SQL_DELETE_CART = "DELETE FROM CART
WHERE ID = :id";
    Map<String, Object> params = new
    HashMap<String, Object>();
}
```

```
        params.put("id", id);
        jdbcTemplate.update(SQL_DELETE_CART_ITEM,
params);
        jdbcTemplate.update(SQL_DELETE_CART, params);
    }
    @Override
    public void addItem(String cartId, String
productId) {
        String SQL=null;
        Cart cart = null;
        cart = read(cartId);
        if(cart ==null) {
            CartItemDto newCartItemDto = new
CartItemDto();
            newCartItemDto.setId(cartId+productId);
            newCartItemDto.setProductId(productId);
            newCartItemDto.setQuantity(1);
            CartDto newCartDto = new CartDto(cartId);
            newCartDto.addCartItem(newCartItemDto);
            create(newCartDto);
            return;
        }
        Map<String, Object> cartItemsParams = new
HashMap<String, Object>();

        if(cart.getItemByProductId(productId) == null) {
            SQL = "INSERT INTO CART_ITEM (ID,
PRODUCT_ID, CART_ID, QUANTITY) VALUES (:id,
:productId, :cartId, :quantity)";
            cartItemsParams.put("id", cartId+productId);
            cartItemsParams.put("quantity", 1);
        } else {
            SQL = "UPDATE CART_ITEM SET QUANTITY =
:quantity WHERE CART_ID = :cartId AND PRODUCT_ID =
:productId";
            CartItem existingItem =
cart.getItemByProductId(productId);
            cartItemsParams.put("id",
existingItem.getId());
            cartItemsParams.put("quantity",
existingItem.getQuantity()+1);
        }
        cartItemsParams.put("productId", productId);
        cartItemsParams.put("cartId", cartId);
        jdbcTemplate.update(SQL, cartItemsParams);
    }

    @Override
```

```
        public void removeItem(String cartId, String
productId) {
            String SQL_DELETE_CART_ITEM = "DELETE FROM
CART_ITEM WHERE PRODUCT_ID = :productId AND CART_ID =
:id";
            Map<String, Object> params = new
HashMap<String, Object>();
            params.put("id", cartId);
            params.put("productId", productId);
            jdbcTemplate.update(SQL_DELETE_CART_ITEM,
params);
        }
    }
```

11. Create an interface named `CartService` under the package `com.packt.webstore.service` in the source folder `src/main/java` and add the following method declarations into it:

```
package com.packt.webstore.service;
import com.packt.webstore.domain.Cart;
import com.packt.webstore.dto.CartDto;

public interface CartService {
    void create(CartDto cartDto);
    Cart read(String cartId);
    void update(String cartId, CartDto cartDto);
    void delete(String id);

    void addItem(String cartId, String productId);

    void removeItem(String cartId, String productId);
}
```

12. Create an implementation class named `CartServiceImpl` for the preceding interface under the package `com.packt.webstore.service.impl` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.service.impl;

import org.springframework.beans.factory.
annotation.Autowired;
import org.springframework.stereotype.Service;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.domain
.repository.CartRepository;
import com.packt.webstore.dto.CartDto;
```

```
import com.packt.webstore.service.CartService;

@Service
public class CartServiceImpl implements CartService{
    @Autowired
    private CartRepository cartRepository;

    public void create(CartDto cartDto) {
        cartRepository.create(cartDto);
    }

    @Override
    public Cart read(String id) {
        return cartRepository.read(id);
    }

    @Override
    public void update(String id, CartDto cartDto) {
        cartRepository.update(id, cartDto);
    }

    @Override
    public void delete(String id) {
        cartRepository.delete(id);
    }

    @Override
    public void addItem(String cartId, String
productId) {
        cartRepository.addItem(cartId, productId);
    }

    @Override
    public void removeItem(String cartId, String
productId) {
        cartRepository.removeItem(cartId, productId);
    }
}
```

13. Now create a class named `CartRestController` under the package `com.packt.webstore.controller` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.controller;

import javax.servlet.http.HttpSession;

import org.springframework.beans.factory
```

```
.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind
.annotation.PathVariable;
import org.springframework.web.bind
.annotation.RequestBody;
import org.springframework.web.bind
.annotation.RequestMapping;
import org.springframework.web.bind
.annotation.RequestMethod;
import org.springframework.web.bind
.annotation.ResponseStatus;
import org.springframework.web.bind
.annotation.RestController;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.dto.CartDto;
import com.packt.webstore.service.CartService;

@RestController
@RequestMapping(value = "rest/cart")
public class CartRestController {

    @Autowired
    private CartService cartService;
    @RequestMapping(method = RequestMethod.POST)
    @ResponseStatus(value = HttpStatus.CREATED)
    public void create(@RequestBody CartDto cartDto) {
        cartService.create(cartDto);
    }

    @RequestMapping(value =("/{cartId}", method =
RequestMethod.GET)
    public Cart read(@PathVariable(value = "cartId")
String cartId) {
        return cartService.read(cartId);
    }

    @RequestMapping(value =("/{cartId}", method =
RequestMethod.PUT)
    @ResponseStatus(value = HttpStatus.OK)
    public void update(@PathVariable(value = "cartId")
String cartId, @RequestBody CartDto cartDto) {
        cartDto.setId(cartId);
        cartService.update(cartId, cartDto);
    }

    @RequestMapping(value =("/{cartId}", method =
```

```
RequestMethod.DELETE)
    @ResponseStatus(value = HttpStatus.OK)
    public void delete(@PathVariable(value = "cartId")
String cartId) {
        cartService.delete(cartId);
    }
    @RequestMapping(value = "/add/{productId}", method
= RequestMethod.PUT)
    @ResponseStatus(value = HttpStatus.OK)
    public void addItem(@PathVariable String
productId, HttpSession session) {
        cartService.addItem(session.getId(), productId);
    }
    @RequestMapping(value = "/remove/{productId}",
method = RequestMethod.PUT)
    @ResponseStatus(value = HttpStatus.OK)
    public void removeItem(@PathVariable String
productId, HttpSession session) {
        cartService.removeItem(session.getId(), productId);
    }
}
```

14. Now run our webstore project from the STS.

What just happened?

Okay, in order to store cart-related information, first of all we need the database tables to store cart-related information, which is why we are creating `CART` and `CART_ITEM` tables in steps 1 and 2.

In steps 3 and 4, we have just created two domain classes called `CartItem` and `Cart` to hold the information about the shopping cart. The `CartItem` class just represents a single item in a shopping cart and it holds information such as the `product`, `quantity`, and the `totalPrice`. Similarly, the `Cart` represents the whole shopping cart itself; a `Cart` can have a collection of `cartItems` and `grandTotal`. Similarly, in steps 5 and 6 we have created two more data transfer objects called `CartItemDto` and `CartDto` to carry data between the REST client and our backend services.

In steps 7 and 8 we created a class called `CartItemMapper` and `CartMapper`, which we are going to use to map the database records to the `CartItem` and `Cart` domain object in the repository classes.

In steps 9 to 10, we just created a repository layer to manage `CartDto` objects. In the `CartRepository` interface, we have defined six methods to take care of CRUD operations (Create, Read, Update, and Delete) on the `CART_ITEM` table. The `InMemoryCartRepository` is just an implementation of the `CartRepository` interface.

Similarly, from steps 11 to 12, we have created the service layer for `Cart` objects. The `CartService` interface has the same methods of the `CartRepository` interface. The `CartServiceImpl` class just internally uses `InMemoryCartRepository` to carry out all the CRUD operations.

Step 13 is very crucial in the whole sequence because in that step we have created our REST-style Controller to handle all `Cart`-related REST web services. The `CartRestController` class mainly has six methods to handle web requests for CRUD operations on `Cart` objects, namely `create`, `read`, `update`, and `delete`, and two more methods, `addItem` and `removeItem`, to handle adding and removing a `CartItem` from the `Cart` object. We will explore the first four CRUD methods in greater details.

You will see, on the surface, the `CartRestController` class is just like any other normal Spring MVC Controller, because it has the same `@RequestMapping` annotations. What makes it special enough to become a REST-style Controller is the `@RestController` and `@RequestBody` annotations. See the following Controller method:

```
@RequestMapping(value =("/{cartId}"), method = RequestMethod.GET)
public Cart read(@PathVariable(value = "cartId") String cartId) {
    return cartService.read(cartId);
}
```

Usually every Controller method is used to return a View name, so that the dispatcher servlet can find the appropriate View file and dispatch that View file to the client, but here we have returned the object (`Cart`) itself. Instead of putting the object into the Model, we have returned the object itself. Why? Because we want to return the state of the object in JSON format. Remember, REST-based web services should return data in JSON or XML format and the client can use the data however they want; they may render it to an HTML page, or they may send it to some external system as it is, as raw JSON/XML data.

Okay, let's come to the point. The `read` Controller method is just returning `Cart` Java objects; how come this Java object is being converted into JSON or XML format? This is where the `@RestController` annotation comes in. The `@RestController` annotation instructs Spring to convert all Java objects that are returned from the request-mapping methods into JSON/XML format and send a response in the body of the HTTP response.

Similarly, when you send an HTTP request to a Controller method with JSON/XML data in it, the `@RequestBody` annotation instructs Spring to convert it into the corresponding Java object. That's why the `create` method has a `cartDto` parameter annotated with a `@RequestBody` annotation.

If you closely observe the `@RequestMapping` annotation of all those six CRUD methods, you will end up with the following table:

URL	HTTP method	Description
<code>http://localhost:8080/webstore/rest/cart</code>	POST	Creates a new cart
<code>http://localhost:8080/webstore/rest/cart/1234</code>	GET	Retrieves cart with ID = 1234
<code>http://localhost:8080/webstore/rest/cart/1234</code>	PUT	Updates cart with ID = 1234
<code>http://localhost:8080/webstore/rest/cart/1234</code>	DELETE	Deletes cart with ID = 1234
<code>http://localhost:8080/webstore/rest/cart/add/P1234</code>	PUT	Adds the product with ID P1234 to the cart under session
<code>http://localhost:8080/webstore/rest/cart/remove/P1234</code>	PUT	Removes the product with ID P1234 to the cart under session

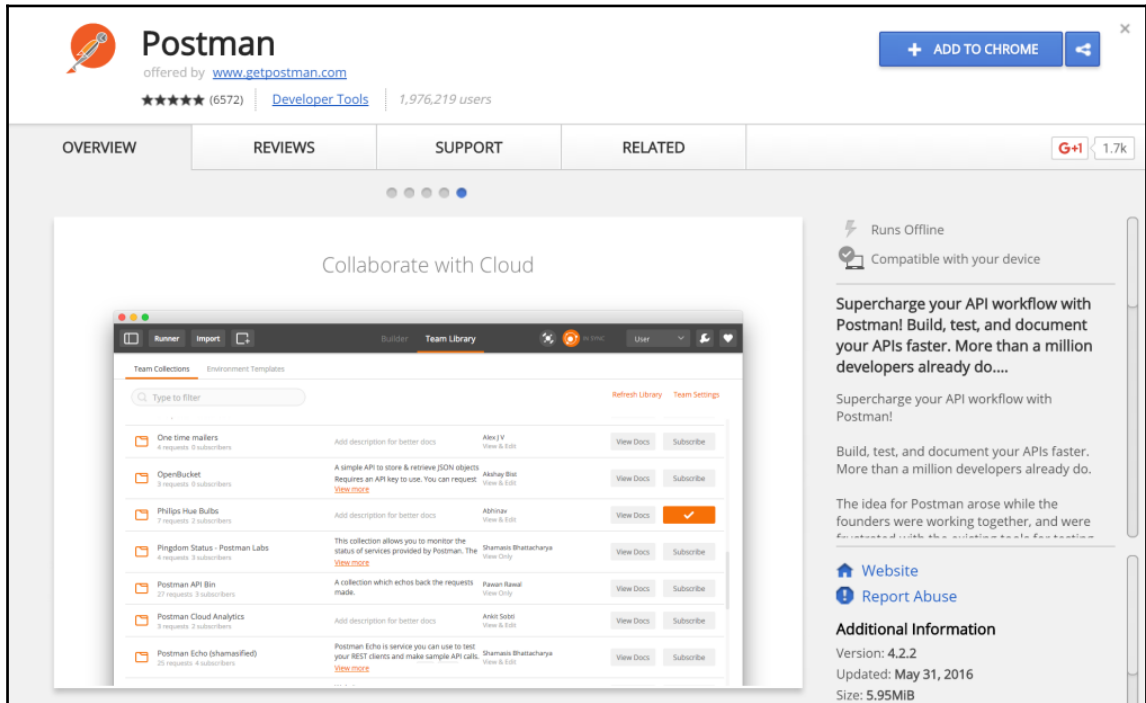
Though the request-mapping URL is more or less the same, based on the HTTP method, we are performing different operations. For example, if you send a GET request to the URL `http://localhost:8080/webstore/rest/cart/1234`, the `read` Controller method would get executed and the `Cart` object with ID 1234 will get returned in JSON format. Similarly, if you send a PUT or DELETE request to the same URL, the `update` or `delete` Controller method would get called correspondingly.

In addition to those four CRUD request-mapping methods, we have two more request-mapping methods that take care of adding and removing a `CartItem` from the `Cart` object. These methods are considered update methods, which is why both `addItem` and `removeItem` methods have PUT as a request method type in their `@RequestMapping` annotation. For instance, if you send a PUT request to the URL `http://localhost:8080/webstore/rest/cart/add/P1236`, a product with a product id P1236 will be added to the `Cart` object. Similarly, if you send a PUT request to the URL `http://localhost:8080/webstore/rest/cart/remove/P1236`, a product with P1236 will be removed from the `Cart` object.

Time for action – consuming REST web services

Okay, we have created our REST-style Controller, which can serve some REST-based web requests, but we have not seen our `CartRestController` in action. Using the standard browser we can only send GET or POST requests; in order to send a PUT or DELETE request we need a special tool. There are plenty of HTTP client tools available to send requests, let's use one such tool called **Postman** to test our `CartRestController`. Postman is a Google Chrome browser extension, so better install Google Chrome in your system before downloading the Postman HTTP client:

1. Go to the Postman download page at `http://www.getpostman.com/` from your Google Chrome browser and click **Chrome App**.
2. You will be taken to the Chrome webstore page; click the + **ADD TO CHROME** button to install the Postman tool in your browser:

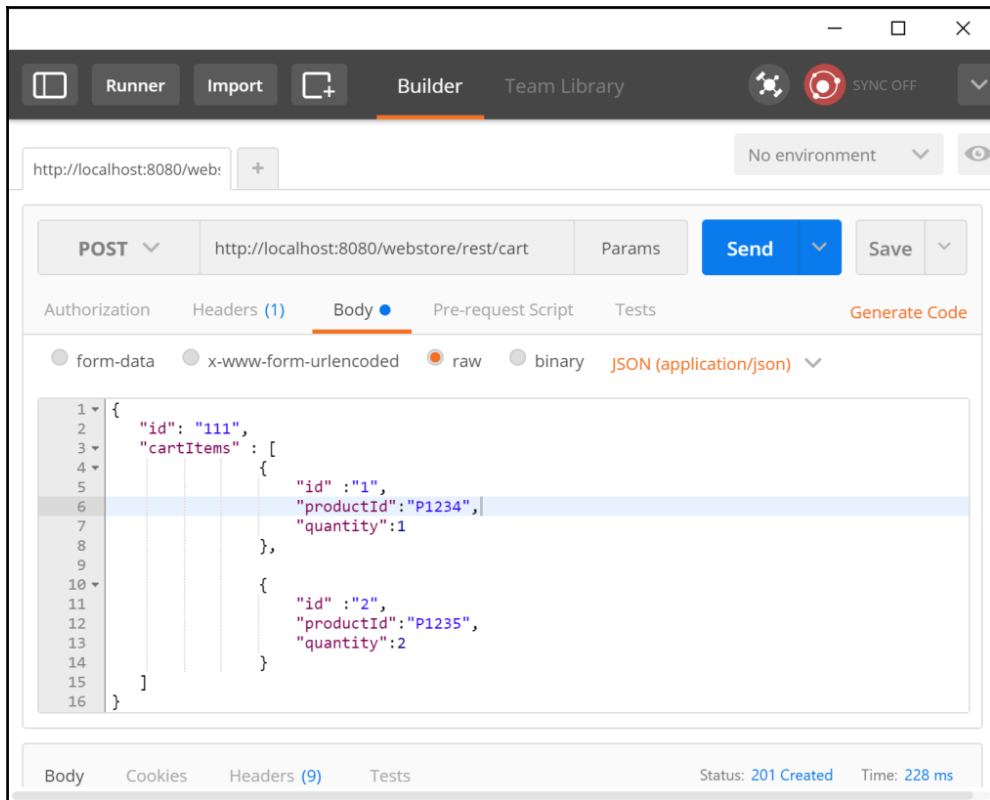


Postman – HTTP client app installing

3. Now a Google login page will appear to ask you to log in. Log in using your Google account.
4. A confirmation dialog will be shown on your screen asking your permission to add the Postman extension to your browser; click the **Add app** button.
5. Now open your Google Chrome browser and enter the URL `chrome://apps/`. A web page will be loaded with all the available apps for your Chrome browser. Just click on the Postman icon to launch the Postman app. Before launching Postman just ensure our webstore project is running.
6. Now, in the Postman app, enter the request URL as `http://localhost:8080/webstore/rest/cart`, the request method as **POST**, the **Body** as **raw** format, and the content type as **JSON(application/json)**.
7. Now enter the following JSON content in the content box and press the **Send** button. An HTTP respond status 201 will be created:

```
{  
  "id": "111",
```

```
"cartItems" : [  
  {  
    "id" : "1",  
    "productId": "P1234",  
    "quantity": 1  
  },  
  {  
    "id" : "2",  
    "productId": "P1235",  
    "quantity": 2  
  }  
]
```



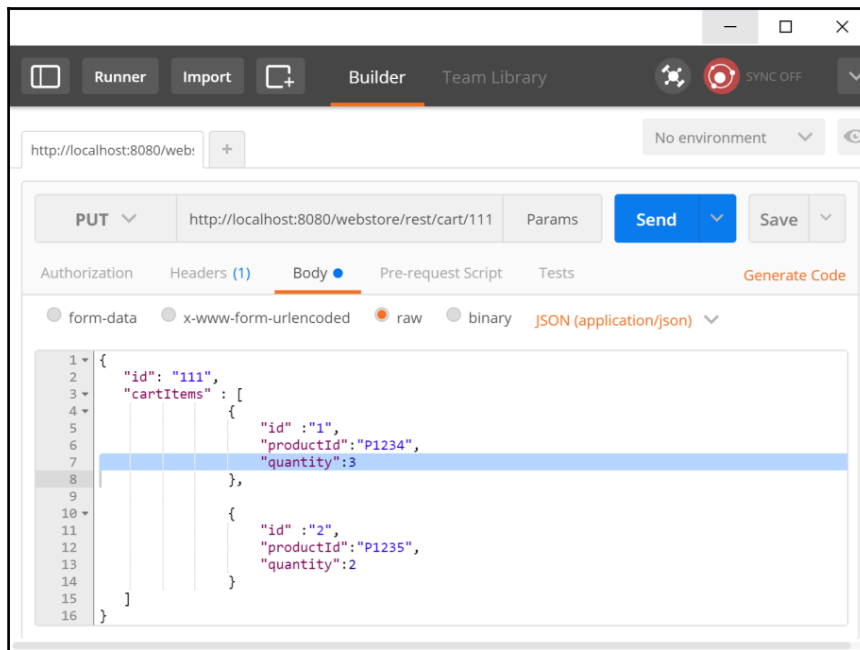
Postman – posting a web request

8. Now, similarly in the Postman app, enter the target URL as `http://localhost:8080/webstore/rest/cart/111` and the request method as **GET**, and press the **Send** button. You will get the following JSON as a response:

```
{
  "id": "111",
  "cartItems": [
    {
      "id": "1",
      "product": {
        "productId": "P1234",
        "name": "iPhone 6s",
        "unitPrice": 500,
        "description": "Apple iPhone 6s smartphone
with 4.00-inch 640x1136 display and 8-megapixel rear
camera",
        "manufacturer": "Apple",
        "category": "Smart Phone",
        "unitsInStock": 450,
        "unitsInOrder": 0,
        "discontinued": false,
        "condition": "New"
      },
      "quantity": 1,
      "totalPrice": 500
    },
    {
      "id": "2",
      "product": {
        "productId": "P1235",
        "name": "Dell Inspiron",
        "unitPrice": 700,
        "description": "Dell Inspiron 14-inch Laptop
(Black) with 3rd Generation Intel Core processors",
        "manufacturer": "Dell",
        "category": "Laptop",
        "unitsInStock": 1000,
        "unitsInOrder": 0,
        "discontinued": false,
        "condition": "New"
      },
      "quantity": 2,
      "totalPrice": 1400
    }
  ],
  "grandTotal": 1900
}
```

```
}
```

- To update the cart in the Postman app, enter the target URL as `http://localhost:8080/webstore/rest/cart/111` and just change the JSON data. For instance, in the content box, just change the `quantity` to 3 for the `P1234` cart item, choose the request method as **PUT** and the content type as **JSON(application/json)**, and send the request to the same URL by pressing the **Send** button:



Postman – posting a PUT web service request.

- To verify whether your changes took place, just repeat step 8; you will see the `totalPrice` increased to 1500 for the cart item with the product ID `P1234`; `grandTotal` will increase accordingly as well.
- Similarly, to delete the cart just enter the `http://localhost:8080/webstore/rest/cart/111` URL in the Postman app, enter the target URL and the request method as **DELETE**, and press the **Send** button. You will get the HTTP status **200 (OK)** as a response. To verify whether the cart got deleted, just repeat step 8; you will get an empty response.

What just happened?

At the start of the chapter, we discussed that most REST-based web services are designed to exchange data in JSON or XML format. This is because the JSON and XML formats are considered universal formats so that any system can easily understand, parse, and interpret that data. In order to test the REST-based web services that we have created, we need a tool that can send different (GET, PUT, POST, or DELETE) types of HTTP requests with JSON data in its request body. Postman is one such tool and is available as a Google Chrome extension.

From steps 1 to 4, we just installed the Postman app in our Google Chrome browser. In steps 6 and 7, we just sent our first REST-based web request to the target URL `http://localhost:8080/webstore/rest/cart` to create a new cart in our webstore application. We did this by sending a POST request with the cart information as JSON data to the target URL. If you notice the following JSON data, it represents a cart with a cart id 111, and it has two product (P1234 and P1235) cart items in it:

```
{
  "id": "111",
  "cartItems" : [
    {
      "id" : "1",
      "productId": "P1234",
      "quantity": 1
    },
    {
      "id" : "2",
      "productId": "P1235",
      "quantity": 2
    }
  ]
}
```

Since we have posted our first cart in our system, to verify whether that cart got stored in our system, we have sent another REST web request in step 8 to get the whole cart information in JSON format. Notice this time the request type is GET and the target URL is `http://localhost:8080/webstore/rest/cart/111`. Remember, we have learned that, in REST-based applications, every resource is identifiable by a URI. Here the URL `http://localhost:8080/webstore/rest/cart/111` represents a cart whose ID is 111. If you send a GET request to the previous URL you will get the cart information as JSON data.

Similarly, you can even update the whole cart by sending an updated JSON data as a PUT request to the same URL, which is what we have done in step 9. In a similar fashion we have sent a DELETE request to the same URL to delete the cart whose ID is 111.

Okay, we have tested or consumed our REST-based web services with the help of the Postman HTTP client tool, which is working quite well. But in a real-world application, most of the time these kinds of REST-based web service are consumed from the frontend with the help of a concept called Ajax. Using a JavaScript library, we can send an Ajax request to the backend. In the next section, we will see what Ajax requests are and how to consume REST-based web services using JavaScript/Ajax libraries.

Handling web services in Ajax

Ajax (Asynchronous JavaScript and XML) is a web development technique used on the client side to create asynchronous web applications. In a typical web application, every time a web request is fired as a response, we get a full web page loaded, but in an Ajax-based web application web pages are updated asynchronously by polling small data with the server behind the scenes. This means that, using Ajax, it is possible to update parts of a web page without reloading the entire web page. With Ajax, web applications can send data to, and retrieve data from, a server asynchronously. The asynchronous aspect of Ajax allows us to write code that can send some requests to a server and handles a server response without reloading the entire web page.

In an Ajax-based application, the `XMLHttpRequest` object is used to exchange data asynchronously with the server, whereas XML or JSON is often used as the format for transferring data. The “X” in AJAX stands for XML, but JSON is used instead of XML nowadays because of its simplicity, and it uses fewer characters to represent the same data compared to XML. So it can reduce the bandwidth requirements over the network to make the data transfer speed faster.

Time for action – consuming REST web services via Ajax

Okay, we have implemented some REST-based web services that can manage the shopping cart in our application, but we need a frontend that can facilitate the end user to manage a shopping cart visually. So let's consume those web services via Ajax in the frontend to manage the shopping cart. In order to consume REST web services via Ajax, perform the following steps:

1. Add a JavaScript file named `controllers.js` under the directory `/src/main/webapp/resources/js/`, add the following code snippets into it, and save it:

```
var cartApp = angular.module('cartApp', []);

cartApp.controller('cartCtrl', function($scope,
$http) {

    $scope.refreshCart = function(cartId) {
        $http.get('/webstore/rest/cart/' +
$scope.cartId)
            .success(function(data) {
                $scope.cart = data;
            });
    };

    $scope.clearCart = function() {
        $http.delete('/webstore/rest/cart/' +
$scope.cartId)
            .success(function(data) {
                $scope.refreshCart($scope.cartId);
            });
    };

    $scope.initCartId = function(cartId) {
        $scope.cartId = cartId;
        $scope.refreshCart($scope.cartId);
    };

    $scope.addToCart = function(productId) {
        $http.put('/webstore/rest/cart/add/' +
productId)
            .success(function(data) {
                alert("Product Successfully added to
the Cart!");
            });
    };

    $scope.removeFromCart = function(productId) {
        $http.put('/webstore/rest/cart/remove/' +
productId)
            .success(function(data) {
                $scope.refreshCart($scope.cartId);
            });
    };
});
```

2. Now create a class named `CartController` under the package `com.packt.webstore.controller` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.controller;

import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind
    .annotation.PathVariable;
import org.springframework.web.bind
    .annotation.RequestMapping;
import org.springframework.web.bind
    .annotation.RequestMethod;

@Controller
@RequestMapping(value = "/cart")
public class CartController {

    @RequestMapping
    public String get(HttpServletRequest request) {
        return
            "redirect:/cart/"+request.getSession(true).getId();
    }

    @RequestMapping(value = "/{cartId}", method =
        RequestMethod.GET)
    public String getCart(@PathVariable(value =
        "cartId") String cartId, Model model) {
        model.addAttribute("cartId", cartId);
        return "cart";
    }
}
```

3. Add one more JSP View file named `cart.jsp` under the directory `src/main/webapp/WEB-INF/views/`, add the following code snippets into it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
```

```
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/
bootstrap/3.0.0/css/bootstrap.min.css">

<script src="https://ajax.googleapis.com/ajax
/libs/angularjs/1.5.1/angular.min.js"></script>

<script src="/webstore/resources/js/
controllers.js"></script>

<title>Cart</title>
</head>
<body>
  <section>
    <div class="jumbotron">
      <div class="container">
        <h1>Cart</h1>
        <p>All the selected products in your
cart</p>
      </div>
    </div>
  </section>

  <section class="container" ng-app="cartApp">
    <div ng-controller="cartCtrl" ng-
init="initCartId('{{cartId}}')">

      <div>
        <a class="btn btn-danger pull-left"
ng-click="clearCart()"> <span
class="glyphicon glyphicon-remove-
sign"></span> Clear Cart
        </a> <a href="#" class="btn btn-success
pull-right"> <span
class="glyphicon glyphicon-shopping-cart
glyphicon"></span> Check out
        </a>
      </div>
      <table class="table table-hover">
        <tr>
          <th>Product</th>
          <th>Unit price</th>
```

```
        <th>Qauntity</th>
        <th>Price</th>
        <th>Action</th>
    </tr>
    <tr ng-repeat="item in cart.cartItems">
        <td>{{item.product.productId}}-
    {{item.product.name}}</td>
        <td>{{item.product.unitPrice}}</td>
        <td>{{item.quantity}}</td>
        <td>{{item.totalPrice}}</td>
        <td><a href="#" class="label label-
danger" ng-
click="removeFromCart(item.product.productId)"> <span
        class="glyphicon glyphicon-
remove" /></span> Remove
        </a></td>
    </tr>
    <tr>
        <th></th>
        <th></th>
        <th>Grand Total</th>
        <th>{{cart.grandTotal}}</th>
        <th></th>
    </tr>
</table>
    <a href="<spring:url
value="/market/products" />" class="btn btn-default">
        <span class="glyphicon-hand-left
glyphicon"></span> Continue shopping
    </a>
</div>
</section>
</body>
</html>
```

4. Open `product.jsp` from `src/main/webapp/WEB-INF/views/` and add the following AngularJS-related script links in the head section as follows:

```
<script src="https://ajax.googleapis.com/ajax
/libs/angularjs/1.5.5/angular.min.js"></script>
```

5. Similarly, add more script links to our `controller.js` as follows:

```
<script src="/webstore/resources  
/js/controllers.js"></script>
```

6. Now add the `ng-click` AngularJS directive to the `Order Now` `<a>` tag as follows:

```
<a href="#" class="btn btn-warning btn-large" ng-  
click="addToCart ('${product.productId} ')">  
<span class="glyphicon-shopping-cart  
glyphicon"></span> Order Now  
</a>
```

7. Finally, add one more `<a>` tag beside the `Order Now` `<a>` tag, which will show the **View cart** button and save the `product.jsp`:

```
<a href="<spring:url value="/cart" />" class="btn  
btn-default">  
  <span class="glyphicon-hand-right  
glyphicon"></span> View Cart  
</a>
```

8. Now add the `ng-app` AngularJS directive to the `Order Now` `<section>` tag as follows:

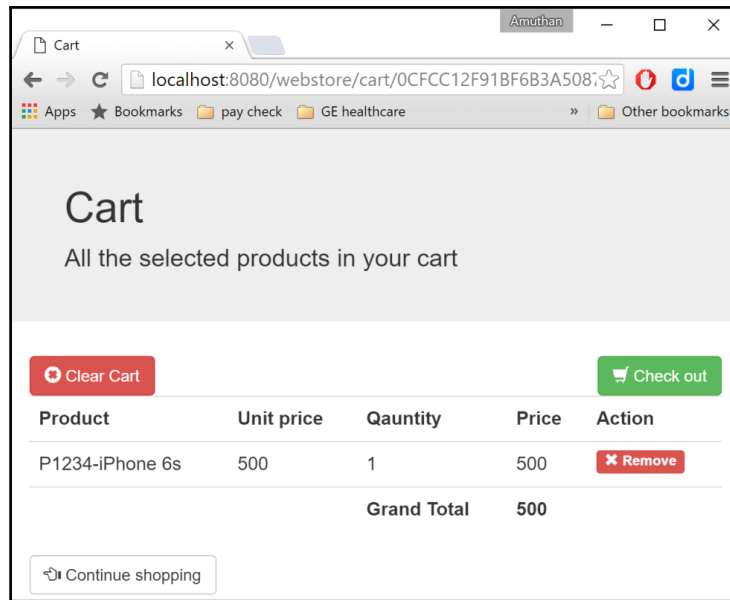
```
<section class="container" ng-app="cartApp">
```

9. Now add the `ng-controller` AngularJS directive to the surrounding `<p>` tag of the `Order Now` link as follows:

```
<p ng-controller="cartCtrl">
```

10. Now run our application and enter the URL `http://localhost:8080/webstore/market/product?id=P1234`. You should be able to see the product detail page of a product whose ID is `P1234`.
11. Now click on the **Order Now** button; an alert message will display, saying **Product successfully added to the cart!!**.

12. Now click on the **View Cart** button; you will see a web page displaying a shopping cart page, as shown in the following screenshot:



Shopping cart page

What just happened?

There are plenty of JavaScript frameworks available to send an Ajax request to the server; we decided to use AngularJS (<https://angularjs.org/>) as our frontend JavaScript library to send Ajax requests. AngularJS is more or less like a frontend MVC framework, but also has the concepts of Model, View, Controller and more. The only difference is that it is designed to work in the frontend using JavaScript.

In step 1, we just created our AngularJS-based Controller called `controllers.js` in `/src/main/webapp/resources/js/`. Remember, we purposely put this file under the `resources` directory because from the client side we want to access this file as a static resource; we don't want to go through Spring MVC Controllers in order to get this file.

Okay, coming to the point, what have we written in `controllers.js`? We have written five frontend Controller methods, namely `refreshCart`, `clearCart`, `initCartId`, `addToCart`, and `removeFromCart`. These methods are used to communicate with the server using Ajax calls. For example, consider the following Controller method:

```
$scope.refreshCart = function(cartId) {  
    $http.get('/webstore/rest/cart/' + $scope.cartId)  
        .success(function(data) {  
            $scope.cart = data;  
        });  
};
```

Within the `refreshCart` method, using the AngularJS `$http` object, we have sent an HTTP GET request to the URI template `/webstore/rest/cart/' + $scope.cartId`. Based on the value stored in the `$scope.cartId` variable, the actual request will be sent to the REST target URL. For instance, if the `$scope.cartId` contains a value of 111, then a GET request will be sent to the `http://localhost:8080/webstore/rest/cart/111` to get a cart object whose ID is 111 as JSON data. Once we get the cart object as JSON data, we store it in the frontend Angular model using the `$scope` object, as follows:

```
.success(function(data) {  
    $scope.cart = data;  
})
```

Similarly, all other AngularJS Controller methods fire some Ajax web requests to the server, and retrieve or update the cart. For example, the `addToCart` and `removeFromCart` methods are just adding a `cartItem` and removing a `cartItem` from the cart object.

Okay, we have just defined our AngularJS Controller methods, but we have to invoke this method in order to do something useful, which is what we have done in step 2. In step 2, we just defined our regular Spring MVC Controller named `CartController`, which has two request-mapping methods, namely `get` and `getCart`. Whenever a normal web request comes to the URL `http://localhost:8080/webstore/cart`, the `get` method will be invoked, and inside the `get` method we have retrieved the session ID and used it as a cart ID to invoke the `getCart` method. Here we maintained the session ID as the cart ID:

```
@RequestMapping  
public String get(HttpServletRequest request) {  
    return "redirect:/cart/"+request.getSession(true).getId();  
}
```


And within the `getCart` method, we simply stored the `cartId` in the Spring MVC Model and returned a View name as `cart`. We did this kind of setup because we want our application to redirect the request to the correct cart based on the session ID whenever a request comes to the URL `http://localhost:8080/webstore/cart`. Okay, since we have returned a View name as `cart`, our dispatcher servlet would definitely look for a View file called `cart.jsp`. That is why we have created the `cart.jsp` in step 3.

`cart.jsp` just acts as a template for our shopping cart page. The `cart.jsp` page internally uses the AngularJS Controller's methods that we have created in step 1 to communicate with the server. The `ng-repeat` directive of AngularJS would repeat the HTML table rows dynamically based on the `cartItems` available in the `cart`:

```
<tr ng-repeat="item in cart.cartItems">
    <td>{{item.product.productId}}-{{item.product.name}}</td>
    <td>{{item.product.unitPrice}}</td>
    <td>{{item.quantity}}</td>
    <td>{{item.totalPrice}}</td>
    <td><a href="#" class="label label-danger" ng-
click="removeFromCart(item.product.productId)">
<span class="glyphicon glyphicon-remove" /></span> Remove
    </td>
</tr>
```

And the `ng-click` directive from the remove `<a>` tag would call the `removeFromCart` Controller method. Similarly, to add a `cartItem` to the cart, in `product.jsp` we have added another `ng-click` directive as follows in step 6 to invoke the `addToCart` method:

```
<a href="#" class="btn btn-warning btn-large" ng-
click="addToCart('${product.productId}')">
<span class="glyphicon-shopping-cart glyphicon"></span> Order Now
</a>
```

So that's it, we have done everything to roll out our shopping cart in our application. After running our application, we can access our shopping cart under the URL `http://localhost:8080/webstore/cart` and can even add products to the cart from each product detail page as well.

Summary

In this chapter, we learned how to develop REST-based web services using Spring MVC and we also learned how to test those web services using the Postman HTTP client tool. We also covered the basic concepts of HTTP verbs and understood how they are related to standard CRUD operations. Finally, we learned how to use the AngularJS JavaScript framework to send Ajax requests to our server.

In the next chapter, we will see how to integrate the Spring Web Flow framework with Spring MVC.

10

Float Your Application with Web Flow

*When it comes to web application development, reusability and maintenance are two important factors that need to be considered. **Spring Web Flow (SWF)** is an independent framework that facilitates the development of highly configurable and maintainable flow-based web applications.*

In this chapter, we are going to see how to incorporate the Spring Web Flow framework within a Spring MVC application. Spring Web Flow facilitates the development of stateful web applications with controlled navigation flow. After finishing this chapter, you will have an idea about developing flow-based applications using Spring Web Flow.

Working with Spring Web Flow

Spring Web Flow allows us to develop flow-based web applications easily. A flow in a web application encapsulates a series of steps that guides the user through the execution of a business task, such as checking in to a hotel, applying for a job, checking out a shopping cart, and so on. Usually, a flow will have clear start and end points, include multiple HTTP requests/responses, and the user must go through a set of screens in a specific order to complete the flow.

In all our previous chapters, the responsibility for defining a page flow specifically lies with controllers, and we weaved the page flows into individual Controllers and Views; for instance, we usually mapped a web request to a controller, and the controller is the one who decides which logical View to return as a response.

This is simple to understand and sufficient for straightforward page flows, but when web applications get more and more complex in terms of user interaction flows, maintaining a large and complex page flow becomes a nightmare.

If you are going to develop such complex flow-based applications, then SWF is your trusty companion. SWF allows you to define and execute **user interface (UI)** flows within your web application. Without further ado, let's dive straight into Spring Web Flow by defining some page flows in our project.

It is nice that we have implemented a shopping cart in our previous chapter, but it is of no use if we do not provide a checkout facility to finish shopping and perform order processing. Let's do that in two phases. Firstly, we need to create the required backend services, domain objects, and repository implementation, in order to perform order processing (here strictly no web flow related stuff is involved, it's just a supportive backend service that can be used later by web flow definitions in order to complete the checkout process). Secondly, we need to define the actual Spring Web Flow definition, which can use our backend services in order to execute the flow definition. There, we will do the actual web flow configuration and definition.

Time for action – implementing the order processing service

We will start by implementing our order processing backend service first. We proceed as follows:

1. Add the following schema definitions for the `CART` and `CART_ITEM` tables in `create-table.sql`; you can find `create-table.sql` under the folder `src/main/resources/db/sql/`:

```
CREATE TABLE ADDRESS (
    ID INTEGER IDENTITY PRIMARY KEY,
    DOOR_NO VARCHAR(25),
    STREET_NAME VARCHAR(25),
    AREA_NAME VARCHAR(25),
    STATE VARCHAR(25),
    COUNTRY VARCHAR(25),
    ZIP VARCHAR(25),
);

CREATE TABLE CUSTOMER (
    ID INTEGER IDENTITY PRIMARY KEY,
    NAME VARCHAR(25),
    PHONE_NUMBER VARCHAR(25),
```

```
BILLING_ADDRESS_ID INTEGER FOREIGN KEY REFERENCES ADDRESS(ID),
);

CREATE TABLE SHIPPING_DETAIL (
    ID INTEGER IDENTITY PRIMARY KEY,
    NAME VARCHAR(25),
    SHIPPING_DATE VARCHAR(25),
    SHIPPING_ADDRESS_ID INTEGER FOREIGN KEY REFERENCES ADDRESS(ID),
);

CREATE TABLE ORDERS (
    ID INTEGER GENERATED BY DEFAULT AS IDENTITY (START WITH
    1000, INCREMENT BY 1) PRIMARY KEY,
    CART_ID VARCHAR(50) FOREIGN KEY REFERENCES CART(ID),
    CUSTOMER_ID INTEGER FOREIGN KEY REFERENCES CUSTOMER(ID),
    SHIPPING_DETAIL_ID INTEGER FOREIGN KEY REFERENCES
    SHIPPING_DETAIL(ID),
);
```

2. Also add the following drop table command as the first line in create-table.sql:

```
DROP TABLE ORDERS IF EXISTS;
DROP TABLE CUSTOMER IF EXISTS;
DROP TABLE SHIPPING_DETAIL IF EXISTS;
DROP TABLE ADDRESS IF EXISTS;
```

3. Create a class named `Address` under the `com.packt.webstore.domain` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.domain;

import java.io.Serializable;

public class Address implements Serializable{

    private static final long serialVersionUID = -530086768384258062L;
    private Long id;
    private String doorNo;
    private String streetName;
    private String areaName;
    private String state;
    private String country;
    private String zipCode;
    public Long getId() {
        return id;
    }
}
```

```
public void setId(Long id) {
    this.id = id;
}
public String getDoorNo() {
    return doorNo;
}
public void setDoorNo(String doorNo) {
    this.doorNo = doorNo;
}
public String getStreetName() {
    return streetName;
}
public void setStreetName(String streetName) {
    this.streetName = streetName;
}
public String getAreaName() {
    return areaName;
}
public void setAreaName(String areaName) {
    this.areaName = areaName;
}
public String getState() {
    return state;
}
public void setState(String state) {
    this.state = state;
}
public String getCountry() {
    return country;
}
public void setCountry(String country) {
    this.country = country;
}
public String getZipCode() {
    return zipCode;
}
public void setZipCode(String zipCode) {
    this.zipCode = zipCode;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Address other = (Address) obj;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    return true;
}
}
```

4. Create another class named `Customer` under the same package, and add the following code to it:

```
package com.packt.webstore.domain;

import java.io.Serializable;

public class Customer implements Serializable{

    private static final long serialVersionUID = 2284040482222162898L;
    private Long customerId;
    private String name;
    private Address billingAddress;
    private String phoneNumber;
    public Customer() {
        super();
        this.billingAddress = new Address();
    }
    public Customer(Long customerId, String name) {
        this();
        this.customerId = customerId;
        this.name = name;
    }

    public Long getCustomerId() {
        return customerId;
    }

    public void setCustomerId(long customerId) {
        this.customerId = customerId;
    }
}
```

```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Address getBillingAddress() {
        return billingAddress;
    }

    public void setBillingAddress(Address billingAddress) {
        this.billingAddress = billingAddress;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public static long getSerialversionuid() {
        return serialVersionUID;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((customerId == null) ? 0 :
            customerId.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Customer other = (Customer) obj;
```



```
        if (customerId == null) {
            if (other.customerId != null)
                return false;
        } else if (!customerId.equals(other.customerId))
            return false;
        return true;
    }
}
```

5. Create one more domain class named `ShippingDetail` under the same package, and add the following code to it:

```
package com.packt.webstore.domain;

import java.io.Serializable;
import java.util.Date;
import org.springframework.format.annotation.DateTimeFormat;

public class ShippingDetail implements Serializable{

    private static final long serialVersionUID = 6350930334140807514L;
    private Long id;
    private String name;
    @DateTimeFormat(pattern = "dd/MM/yyyy")
    private Date shippingDate;
    private Address shippingAddress;
    public Long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public ShippingDetail() {
        this.shippingAddress = new Address();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getShippingDate() {
        return shippingDate;
    }
}
```

```
public void setShippingDate(Date shippingDate) {
    this.shippingDate = shippingDate;
}

public Address getShippingAddress() {
    return shippingAddress;
}

public void setShippingAddress(Address shippingAddress) {
    this.shippingAddress = shippingAddress;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    ShippingDetail other = (ShippingDetail) obj;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    return true;
}
}
```

6. Similarly, create our final domain class, named `Order`, under the same package, and add the following code to it:

```
package com.packt.webstore.domain;

import java.io.Serializable;

public class Order implements Serializable{

    private static final long serialVersionUID =
```

```
-3560539622417210365L;
private Long orderId;
private Cart cart;
private Customer customer;
private ShippingDetail shippingDetail;
public Order() {
    this.customer = new Customer();
    this.shippingDetail = new ShippingDetail();
}

public Long getOrderId() {
    return orderId;
}

public void setOrderId(Long orderId) {
    this.orderId = orderId;
}

public Cart getCart() {
    return cart;
}

public void setCart(Cart cart) {
    this.cart = cart;
}

public Customer getCustomer() {
    return customer;
}

public void setCustomer(Customer customer) {
    this.customer = customer;
}

public ShippingDetail getShippingDetail() {
    return shippingDetail;
}

public void setShippingDetail(ShippingDetail shippingDetail) {
    this.shippingDetail = shippingDetail;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((orderId == null) ? 0 :
        orderId.hashCode());
}
```

```
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Order other = (Order) obj;
        if (orderId == null) {
            if (other.orderId != null)
                return false;
        } else if (!orderId.equals(other.orderId))
            return false;
        return true;
    }
}
```

7. Next, create an exception class called `InvalidCartException` under the `com.packt.webstore.exception` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.exception;

public class InvalidCartException extends RuntimeException {

    private static final long serialVersionUID =
        -519204156303358491L;
    private String cartId;

    public InvalidCartException(String cartId) {
        this.cartId = cartId;
    }

    public String getCartId() {
        return cartId;
    }
}
```

8. Open the `CartRepository` interface from the `com.packt.webstore.domain.repository` package, and add one more method declaration to it as follows:

```
void clearCart(String cartId);
```

9. Now open the `InMemoryCartRepository` implementation class from the `com.packt.webstore.domain.repository.impl` package and add the following method definition to it:

```
@Override
public void clearCart(String cartId) {
    String SQL_DELETE_CART_ITEM = "DELETE FROM CART_ITEM WHERE CART_ID
= :id";
    Map<String, Object> params = new HashMap<>();
    params.put("id", cartId);
    jdbcTemplate.update(SQL_DELETE_CART_ITEM, params);
}
```

10. Now open the `CartService` interface from the `com.packt.webstore.service` package in the `src/main/java` source folder and add two more method declarations to it as follows:

```
Cart validate(String cartId);
void clearCart(String cartId);
```

11. Next, open the `CartServiceImpl` implementation class from the `com.packt.webstore.service.impl` package in the `src/main/java` source folder, and add the following method implementations to it:

```
@Override
public Cart validate(String cartId) {
    Cart cart = cartRepository.read(cartId);
    if(cart==null || cart.getCartItems().size()==0) {
        throw new InvalidCartException(cartId);
    }
    return cart;
}

@Override
public void clearCart(String cartId) {
    cartRepository.clearCart(cartId);
}
```

12. Next, create an interface named `OrderRepository` under the `com.packt.webstore.domain.repository` package in the `src/main/java` source folder, and add a single method declaration to it as follows:

```
package com.packt.webstore.domain.repository;

import com.packt.webstore.domain.Order;
```

```
public interface OrderRepository {  
    long saveOrder(Order order);  
}
```

13. Create an implementation class called `InMemoryOrderRepository` under the `com.packt.webstore.domain.repository.impl` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.domain.repository.impl;  
  
import java.util.HashMap;  
import java.util.Map;  
  
import org.springframework.beans.factory  
    .annotation.Autowired;  
import org.springframework.jdbc.core.  
    namedparam.MapSqlParameterSource;  
import org.springframework.jdbc.core.  
    namedparam.NamedParameterJdbcTemplate;  
import org.springframework.jdbc.core.  
    .namedparam.SqlParameterSource;  
import org.springframework.jdbc.support.GeneratedKeyHolder;  
import org.springframework.jdbc.support.KeyHolder;  
import org.springframework.stereotype.Repository;  
  
import com.packt.webstore.domain.Address;  
import com.packt.webstore.domain.Customer;  
import com.packt.webstore.domain.Order;  
import com.packt.webstore.domain.ShippingDetail;  
import com.packt.webstore.domain.repository.OrderRepository;  
import com.packt.webstore.service.CartService;  
  
@Repository  
public class InMemoryOrderRepository implements OrderRepository {  
  
    @Autowired  
    private NamedParameterJdbcTemplate jdbcTemplate;  
    @Autowired  
    private CartService cartService;  
  
    @Override  
    public long saveOrder(Order order) {  
        Long customerId = saveCustomer(order.getCustomer());  
        Long shippingDetailId =  
            saveShippingDetail(order.getShippingDetail());  
        order.getCustomer().setCustomerId(customerId);  
        order.getShippingDetail().setId(shippingDetailId);  
    }  
}
```

```
        long createdOrderId = createOrder(order);
        CartService.clearCart(order.getCart().getId());
        return createdOrderId;
    }
    private long saveShippingDetail(ShippingDetail shippingDetail) {
        long addressId =
            saveAddress(shippingDetail.getShippingAddress());
        String SQL = "INSERT INTO
        SHIPPING_DETAIL(NAME,SHIPPING_DATE,SHIPPING_ADDRESS_ID) "
            + "VALUES (:name, :shippingDate, :addressId)";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("name", shippingDetail.getName());
        params.put("shippingDate", shippingDetail.getShippingDate());
        params.put("addressId", addressId);

        SqlParameterSource paramSource = new
        MapSqlParameterSource(params);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcTempleate.update(SQL, paramSource, keyHolder, new
        String[]{"ID"});
        return keyHolder.getKey().longValue();
    }

    private long saveCustomer(Customer customer) {
        long addressId = saveAddress(customer.getBillingAddress());
        String SQL = "INSERT INTO
        CUSTOMER(NAME,PHONE_NUMBER,BILLING_ADDRESS_ID) "
            + "VALUES (:name, :phoneNumber, :addressId)";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("name", customer.getName());
        params.put("phoneNumber", customer.getPhoneNumber());
        params.put("addressId", addressId);

        SqlParameterSource paramSource = new
        MapSqlParameterSource(params);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcTempleate.update(SQL, paramSource, keyHolder, new
        String[]{"ID"});
        return keyHolder.getKey().longValue();
    }

    private long saveAddress(Address address) {
        String SQL = "INSERT INTO
        ADDRESS(DOOR_NO,STREET_NAME,AREA_NAME,STATE,COUNTRY,ZIP) "
            + "VALUES (:doorNo, :streetName, :areaName, :state,
            :country, :zip)";
```

```
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("doorNo", address.getDoorNo());
        params.put("streetName", address.getStreetName());
        params.put("areaName", address.getAreaName());
        params.put("state", address.getState());
        params.put("country", address.getCountry());
        params.put("zip", address.getZipCode());

        SqlParameterSource paramSource = new
        MapSqlParameterSource(params);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcTemplate.update(SQL, paramSource, keyHolder, new
        String[]{"ID"});
        return keyHolder.getKey().longValue();
    }

    private long createOrder(Order order) {

        String SQL = "INSERT INTO
        ORDERS(CART_ID,CUSTOMER_ID,SHIPPING_DETAIL_ID) "
            + "VALUES (:cartId, :customerId, :shippingDetailId)";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("id", order.getOrderid());
        params.put("cartId", order.getCart().getId());
        params.put("customerId", order.getCustomer().getCustomerId());
        params.put("shippingDetailId",
        order.getShippingDetail().getId());

        SqlParameterSource paramSource = new
        MapSqlParameterSource(params);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcTemplate.update(SQL, paramSource, keyHolder, new
        String[]{"ID"});
        return keyHolder.getKey().longValue();
    }
}
```

14. Create an interface named `OrderService` under the `com.packt.webstore.service` package in the `src/main/java` source folder and add the following method declarations to it as follows:

```
package com.packt.webstore.service;

import com.packt.webstore.domain.Order;

public interface OrderService {
```



```
        Long saveOrder(Order order);  
    }  
}
```

15. Create an implementation class named `OrderServiceImpl` for the previous interface under the `com.packt.webstore.service.impl` package in the `src/main/java` source folder and add the following code to it:

```
package com.packt.webstore.service.impl;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import com.packt.webstore.domain.Order;  
import com.packt.webstore.domain.repository.OrderRepository;  
import com.packt.webstore.service.OrderService;  
  
@Service  
public class OrderServiceImpl implements OrderService{  
    @Autowired  
    private OrderRepository orderRepository;  
    @Override  
    public Long saveOrder(Order order) {  
        return orderRepository.saveOrder(order);  
    }  
}
```

What just happened?

I guess what we have done so far must already be familiar to you: we have created some domain classes (`Address`, `Customer`, `ShippingDetail`, and `Order`), an `OrderRepository` interface, and its implementation class, `InMemoryOrderRepositoryImpl`, to store processed `Order` domain objects. And finally, we also created the corresponding `OrderService` interface and its implementation class `OrderServiceImpl`.

On the surface, it looks the same as usual, but there are some minute details that need to be explained. If you notice, all the domain classes that we created from steps 1 to 4 have just implemented the `Serializable` interface; not only that, we have even implemented the `Serializable` interface for other existing domain classes as well, such as `Product`, `CartItem`, and `Cart`. This is because later we are going to use these domain objects in Spring Web Flow, and Spring Web Flow is going to store these domain objects in a session for state management between page flows.

Session data can be saved onto a disk or transferred to other web servers during clustering. So when the session object is re-imported from a disk, Spring Web Flow de-serializes the domain object (that is, the form backing bean) to maintain the state of the page. That's why it is a must to serialize the domain object/form backing bean. Spring Web Flow uses a term called `Snapshot` to mention these states within a session.

The remaining steps, steps 6 to 13, are self-explanatory. We have created the `OrderRepository` and `OrderService` interfaces and their corresponding implementations, `InMemoryOrderRepositoryImpl` and `OrderServiceImpl`. The purpose of these classes is to save the `Order` domain object. The `saveOrder` method from `OrderServiceImpl` just deletes the corresponding `CartItem` objects from `CartRepository`, after successfully saving the `order` domain object. Now we have successfully created all the required backend services and domain objects, in order to kick off our Spring Web Flow configuration and definition.

Time for action – implementing the checkout flow

We will now add Spring Web Flow support to our project and define the checkout flow for our shopping cart:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project.
2. You will be able to see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as `org.springframework.webflow`, **Artifact Id** as `spring-webflow`, **Version** as `2.4.2.RELEASE`, select **Scope** as **compile**, click on the **OK** button, and save `pom.xml`.
4. Create a directory structure `flows/checkout/` under the `src/main/webapp/WEB-INF/` directory, create an XML file called `checkout-flow.xml` in `flows/checkout/`, add the following content into it, and save it:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/
webflow/spring-webflow.xsd">

  <var name="order" class="com.packt.webstore.domain.Order"
  />

  <action-state id="addCartToOrder">
```

```
<evaluate expression="cartServiceImpl.validate
(requestParameters.cartId) "
    result="order.cart" />
<transition to="invalidCartWarning"
    on-
exception="com.packt.webstore.exception
.InvalidCartException" />
    <transition to="collectCustomerInfo" />
</action-state>

<view-state id="collectCustomerInfo"
    view="collectCustomerInfo.jsp" model="order">
    <transition on="customerInfoCollected"
        to="collectShippingDetail" />
</view-state>

<view-state id="collectShippingDetail" model="order">
    <transition on="shippingDetailCollected"
        to="orderConfirmation" />
    <transition on="backToCollectCustomerInfo"
        to="collectCustomerInfo" />
</view-state>

<view-state id="orderConfirmation">
    <transition on="orderConfirmed" to="processOrder" />
    <transition on="backToCollectShippingDetail"
        to="collectShippingDetail" />
</view-state>
<action-state id="processOrder">
    <evaluate expression="orderServiceImpl.saveOrder(order) "
        result="order.orderId"/>
    <transition to="thankCustomer" />
</action-state>
<view-state id="invalidCartWarning">
    <transition to="endState"/>
</view-state>
<view-state id="thankCustomer" model="order">
    <transition to="endState"/>
</view-state>

<end-state id="endState"/>

<end-state id="cancelCheckout" view = "checkOutCancelled.jsp"/>
<global-transitions>
    <transition on = "cancel" to="cancelCheckout" />
</global-transitions>
</flow>
```

5. Now, create a web flow configuration class called `WebFlowConfig` under the `com.packt.webstore.config` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.webflow.config
    .AbstractFlowConfiguration;
import org.springframework.webflow.definition
    .registry.FlowDefinitionRegistry;
import org.springframework.webflow.executor.FlowExecutor;
import org.springframework.webflow.mvc.servlet.FlowHandlerAdapter;
import org.springframework.webflow.mvc.servlet.FlowHandlerMapping;

@Configuration
public class WebFlowConfig extends AbstractFlowConfiguration {

    @Bean
    public FlowDefinitionRegistry flowRegistry() {
        return getFlowDefinitionRegistryBuilder()
            .setBasePath("/WEB-INF/flows")
            .addFlowLocationPattern("/**/*.xml")
            .build();
    }

    @Bean
    public FlowExecutor flowExecutor() {
        return getFlowExecutorBuilder(flowRegistry()).build();
    }

    @Bean
    public FlowHandlerMapping flowHandlerMapping() {
        FlowHandlerMapping handlerMapping = new FlowHandlerMapping();
        handlerMapping.setOrder(-1);
        handlerMapping.setFlowRegistry(flowRegistry());
        return handlerMapping;
    }

    @Bean
    public FlowHandlerAdapter flowHandlerAdapter() {
        FlowHandlerAdapter handlerAdapter = new FlowHandlerAdapter();
        handlerAdapter.setFlowExecutor(flowExecutor());
        handlerAdapter.setSaveOutputToFlashScopeOnRedirect(true);
        return handlerAdapter;
    }
}
```

What just happened?

From steps 1 to 3, we just added the Spring Web Flow dependency to our project through Maven configuration. It will download and configure all the required web flow-related JARs for our project. In step 4, we created our first flow definition file, called `checkout-flow.xml`, under the `/src/main/webapp/WEB-INF/flows/checkout/` directory.

Spring Web Flow uses the flow definition file as a basis for executing the flow. In order to understand what has been written in this file, we need to get a clear idea of some of the basic concepts of Spring Web Flow. We will learn about those concepts in a little bit, and then we will come back to `checkout-flow.xml` to understand it better.

Understanding flow definitions

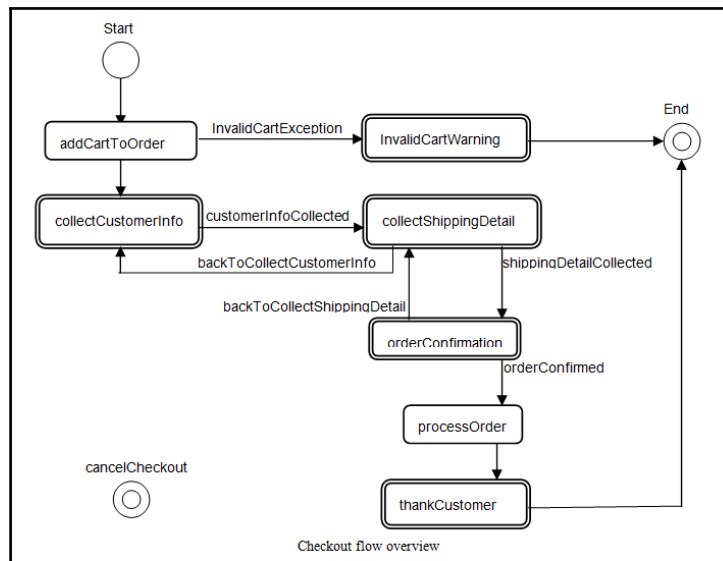
A flow definition is composed of a set of states. Each state will have a unique ID in the flow definition. There are five types of state available in Spring Web Flow:

- **start-state:** Each flow must have a single start state, which helps in creating the initial state of the flow. Note that if the `start-state` is not specified, the very first defined state within the flow definition file becomes the start state.
- **action-state:** A flow can have many action states; an `action-state` executes a particular action. An action normally involves interacting with backend services, such as executing some methods in Spring managed beans; Spring Web Flow uses the **Spring Expression Language** to interact with the backend service beans.
- **view-state:** A `view-state` defines a logical View and Model to interact with the end user. A web flow can have multiple `view-states`. If the `View` attribute is not specified, then the ID of the `view-state` acts as the logical View name.
- **decision-state:** This is used to branch the flow; based on a test condition it routes the transition to the next possible state.
- **subflow-state:** It is an independent flow that can be reused from inside another flow. When an application enters a subflow, the main flow is paused until the subflow completes.
- **end-state:** This state denotes the end of a flow execution. A web flow can have multiple end states; through the `view` attribute of an `end-state`, we can specify a View that will be rendered when its end state is reached.

We have just learned that a flow definition is composed of a set of states, but in order to move from one state to another, we need to define transitions in states. Each state in a web flow (except for the start and end states) defines a number of transitions to move from one state to another. A transition can be triggered by an event signaled by the state.

Understanding checkout flow

Okay, we just got the minimum required introduction to Spring Web Flow concepts; there are plenty of advanced concepts out there to master in Spring Web Flow. We are not going to see all those things, because that itself deserves a separate book. As of now, this is enough to understand the `checkout-flow.xml` flow definition file. But before that, we will provide a quick overview of our checkout flow. The following diagram will give you the overall idea of the checkout flow that we just implemented:



Our checkout flow diagram has a start state and an end state; each rounded rectangle in the diagram defines an action state and each double-line-bordered rounded rectangle defines a view state. Each arrowed line defines transition, and the name associated with it defines the event that causes that particular transition. The `checkout-flow.xml` file just contains this flow in an XML representation.

If you open the `checkout-flow.xml` file, the first tag you encounter within the `<flow>` tag is the `<var>` tag:

```
<var name="order" class="com.packt.webstore.domain.Order" />
```

The `<var>` tag creates a variable in a flow. This variable will be available to all states in a flow, which means we can reference and use this variable inside any state within the flow. In the preceding `<var>` tag, we just created a new instance of the `Order` class and stored it in a variable called `order`.

The next thing we defined within the `checkout-flow.xml` file is the `<action-state>` definition. As we already learned, action states are normally used to invoke backend services, so in the following `<action-state>` definition we have just invoked the `validate` method of the `cartServiceImpl` object and stored the result in the `order.cart` object:

```
<action-state id="addCartToOrder">
    <evaluate expression =
        "cartServiceImpl.validate(requestParameters.cartId)"
        result="order.cart" />
    <transition to="invalidCartWarning" on-exception =
        "com.packt.webstore.exception.InvalidCartException" />

    <transition to="collectCustomerInfo" />
</action-state>
```

As we already defined the `order` variable at the start of the flow, it will be available in every state of this flow. So we have used that variable (`order.cart`) in the `<evaluate>` tag to store the result of this evaluated expression: `cartServiceImpl.validate(requestParameters.cartId)`.

The `validate` method of `cartServiceImpl` tries to read a `cart` object based on the given `cartId`. If it finds a valid `cart` object, then it returns that. Otherwise, it will throw an `InvalidCartException`; in such a case we route the transition to another state whose ID is `invalidCartWarning`:

```
<transition to="invalidCartWarning" on-exception =
    "com.packt.webstore.exception.InvalidCartException" />
```

If no such exception is thrown from the expression evaluation, we naturally transit from the `addCartToOrder` state to the `collectCustomerInfo` state:

```
<transition to="collectCustomerInfo" />
```

If you notice the `collectCustomerInfo` state, it is nothing but a view state in `checkout-flow.xml`. We defined the View that needs to be rendered via the `view` attribute and the Model that needs to be attached via the `model` attribute:

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
```

```
model="order">
  <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

Upon reaching this view state, Spring Web Flow renders the `collectCustomerInfo` View and waits for the user to interact; once the user has entered the customer info details and pressed the **submit** button, it will resume its transition to the `collectShippingDetail` view state. As we already learned, a transition can be triggered via an event, so here the transition to the `collectShippingDetail` state would get triggered when the `customerInfoCollected` event is triggered. How do we fire this event (`customerInfoCollected`) from the View? We will see later in this chapter:

```
<transition on="customerInfoCollected" to="collectShippingDetail" />
```

The next state defined within the checkout flow is `collectShippingDetail`; again this is also a view state, and it has two transitions back and forth: one is to go back to the `collectCustomerInfo` state and the next is to go forward to the `orderConfirmation` state:

```
<view-state id="collectShippingDetail" model="order">
  <transition on="shippingDetailCollected" to="orderConfirmation" />
  <transition on="backToCollectCustomerInfo" to="collectCustomerInfo" />
</view-state>
```

Note that here in the `collectShippingDetail` state, we haven't mentioned the `view` attribute; in that case Spring Web Flow would consider the `id` of the view state to be the View name.

The `orderConfirmation` state definition doesn't need much explanation. It is more like the `collectShippingDetail` view state, where we have furnished all the order-related details and we ask the user to confirm them; upon confirmation, we move to the next state, which is `processOrder`:

```
<view-state id="orderConfirmation">
  <transition on="orderConfirmed" to="processOrder" />
  <transition on="backToCollectShippingDetail" to =
"collectShippingDetail" />
</view-state>
```

Next, the `processOrder` state is an action state that interacts with the `orderServiceImpl` object to save the `order` object. Upon successfully saving the `order` object, it stores the order ID in the flow variable (`order.orderId`) and transits to the next state, which is `thankCustomer`:

```
<action-state id="processOrder">
```



```
<evaluate expression="orderServiceImpl.saveOrder(order)"
result="order.orderId"/>
<transition to="thankCustomer" />
</action-state>
```

The `thankCustomer` state is a view state that simply shows a thank you message with the confirmed order ID to the end user, and transits to the end state:

```
<view-state id="thankCustomer" model="order">
  <transition to="endState"/>
</view-state>
```

In our checkout flow, we have two end states; one is the normal end state where the flow execution arrives naturally after the flow ends, and the other one is the end state when the user presses the **Cancel** button in any of the Views:

```
<end-state id="endState"/>
<end-state id="cancelCheckout" view="checkOutCancelled.jsp"/>
```

Note in the `cancelCheckout` end state, we have specified the name of the landing page via the `view` attribute. The transition to the `cancelCheckout` end state happened through the global transitions configuration:

```
<global-transitions>
  <transition on = "cancel" to="cancelCheckout" />
</global-transitions>
```

A global transition is for sharing some common transitions between states. Rather than repeating the transition definition every time within the state definition, we can define it within one global transition so that that transition will be available implicitly for every state in the flow. In our case, the end user may cancel the checkout process in any state; that's why we have defined the transition to the `cancelCheckout` state in `global-transitions`.

Okay, we have totally understood the checkout flow definition (`checkout-flow.xml`). Now our Spring MVC should read this file during the boot up of our application, so that it can be ready to dispatch any flow-related request to the Spring Web Flow framework. We are able to do this via some web flow configurations, as mentioned in step 5.

In step 5, we have created beans for `FlowExecutor` and the `FlowDefinitionRegistry`. As its name implies, the `flowExecutor` executes a flow based on the given flow definition. The `flowExecutor` gets its flow definition from a `flowDefinitionRegistry` bean. We can configure as many flow definitions in a `flowDefinitionRegistry` as we want.

A `flowDefinitionRegistry` is a collection of flow definitions. When a user enters a flow, the flow executor creates and launches an exclusive flow instance for that user based on the flow definition:

```
@Bean
public FlowDefinitionRegistry flowRegistry() {
    return getFlowDefinitionRegistryBuilder()
        .setBasePath("/WEB-INF/flows")
        .addFlowLocationPattern("/**/*-flow.xml")
        .build();
}
```

In the preceding web flow configuration, we created the `flowDefinitionRegistry` bean, whose base-path is `/WEB-INF/flows`, so we need to put all our flow definitions under the `/WEB-INF/flows` directory in order to get picked up by the `flowDefinitionRegistry`. That's why in step 4 we created our `checkout-flow.xml` under the `src/main/webapp/WEB-INF/flows/checkout/` directory. As I already mentioned, a `flowDefinitionRegistry` can have many flow definitions; each flow definition is identified by its ID within the `flowDefinitionRegistry`. In our case, we have added a single flow definition, whose ID is `checkout` and whose relative location is `/checkout/checkout-flow.xml`.

One important thing to understand before we wind up web flow configuration is the ID of a flow definition forms the relative URL to invoke the flow. By this what I mean is that in order to invoke our checkout flow via a web request, we need to fire a GET request to the `http://localhost:8080/webstore/checkout` URL, because our flow ID is `checkout`. Also, in our flow definition (`checkout-flow.xml`), we haven't configured any start-state, so the first state definition (which is the `addCartToOrderaction-state`) will become the start-state, and `addCartToOrderaction-state`, expecting a `cartId`, should be present in the request parameter of the invoking URL:

```
<action-state id="addCartToOrder">
    <evaluate expression =
"cartServiceImpl.validate(requestParameters.cartId)" result="order.cart" />
    <transition to="invalidCartWarning" on-
exception="com.packt.webstore.exception.InvalidCartException" />
    <transition to="collectCustomerInfo" />
</action-state>
```

So the actual URL that can invoke this flow would be something similar to `http://localhost:8080/webstore/checkout?cartId=55AD1472D4EC`, where the part after the question mark (`cartId=55AD1472D4EC`) is considered as a request parameter.

It is good that we have defined our checkout flow and configured it with the Spring Web Flow, but we need to define two more beans named `flowHandlerMapping` and `flowHandlerAdapter` in our `WebFlowConfig.java`, to dispatch all flow-related requests to the `flowExecutor`. We did that as follows:

```
@Bean
public FlowHandlerMapping flowHandlerMapping() {
    FlowHandlerMapping handlerMapping = new FlowHandlerMapping();
    handlerMapping.setOrder(-1);
    handlerMapping.setFlowRegistry(flowRegistry());
    return handlerMapping;
}

@Bean
public FlowHandlerAdapter flowHandlerAdapter() {
    FlowHandlerAdapter handlerAdapter = new FlowHandlerAdapter();
    handlerAdapter.setFlowExecutor(flowExecutor());
    handlerAdapter.setSaveOutputToFlashScopeOnRedirect(true);
    return handlerAdapter;
}
```

`flowHandlerMapping` creates and configures handler mapping, based on the flow ID for each defined flow from `flowRegistry`. `flowHandlerAdapter` acts as a bridge between the dispatcher servlet and Spring Web Flow, in order to execute the flow instances.

Pop quiz – web flow

Consider the following web flow registry configuration; it has a single flow definition file, namely `validate.xml`. How will you form the URL to invoke the flow?

```
@Bean
public FlowDefinitionRegistry flowRegistry() {
    return getFlowDefinitionRegistryBuilder()
        .setBasePath("/WEB-INF/flows")
        .addFlowLocation("/customer/validate.xml", "validateCustomer")
        .build();
}
```

1. `http://localhost:8080/webstore/customer/validate`
2. `http://localhost:8080/webstore/validate`
3. `http://localhost:8080/webstore/validateCustomer`

Consider the following flow invoking URL:

`http://localhost:8080/webstore/validate?customerId=C1234`

In a flow definition file, how will you retrieve the `customerId` HTTP request parameter?

1. `<evaluate expression = "requestParameters.customerId " result = "customerId" />`
2. `<evaluate expression = "requestParameters(customerId) " result = "customerId" />`
3. `<evaluate expression = "requestParameters[customerId] " result = "customerId" />`

Time for action – creating Views for every view state

We have done everything to roll out our checkout flow, but one last thing is pending: creating all the Views that need to be used in the view states of our checkout flow. In total, we have six view states in our flow definition (`collectCustomerInfo`, `collectShippingDetail`, `orderConfirmation`, `invalidCartWarning`, `thankCustomer`, and `cancelCheckout`), so we need to create six JSP files. Let's create all of them:

1. Create a JSP View file called `collectCustomerInfo.jsp` under the `src/main/webapp/WEB-INF/flows/checkout/` directory, add the following code snippet to it, and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset="utf-8">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Customer</title>
</head>
<body>
<section>
<div class="jumbotron">
<div class="container">
```

```
        <h1>Customer</h1>
        <p>Customer details</p>
    </div>
</div>
</section>
<section class="container">
    <form:form modelAttribute="order.customer" class="form-
    horizontal">
        <fieldset>
            <legend>Customer Details</legend>

            <div class="form-group">
                <label class="control-label col-lg-2"
                for="name">Name</label>
                <div class="col-lg-10">
                    <form:input id="name" path="name"
                    type="text" class="form:input-large" />
                </div>
            </div>

            <div class="form-group">
                <label class="control-label col-lg-2"
                for="doorNo">Door No</label>
                <div class="col-lg-10">
                    <form:input id="doorNo"
                    path="billingAddress.doorNo" type="text"
                    class="form:input-large" />
                </div>
            </div>

            <div class="form-group">
                <label class="control-label col-lg-2"
                for="streetName">Street Name</label>
                <div class="col-lg-10">
                    <form:input id="streetName"
                    path="billingAddress.streetName."
                    type="text"
                    class="form:input-large" />
                </div>
            </div>

            <div class="form-group">
                <label class="control-label col-lg-2"
                for="areaName">Area Name</label>
                <div class="col-lg-10">
                    <form:input id="areaName"
                    path="billingAddress.areaName" type="text"
                    class="form:input-large" />
                </div>
            </div>
        </fieldset>
    </form:form>
</section>
```

```
    </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="state">State</label>
  <div class="col-lg-10">
    <form:input id="state"
      path="billingAddress.state" type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="country">country</label>
  <div class="col-lg-10">
    <form:input id="country"
      path="billingAddress.country" type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="zipCode">Zip Code</label>
  <div class="col-lg-10">
    <form:input id="zipCode"
      path="billingAddress.zipCode" type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="phoneNumber">Phone Number</label>
  <div class="col-lg-10">
    <form:input id="phoneNumber"
      path="phoneNumber" type="text"
      class="form:input-large" />
  </div>
</div>

<input type="hidden" name="_flowExecutionKey"
  value="{flowExecutionKey}"/>
<div class="form-group">
  <div class="col-lg-offset-2 col-lg-10">
    <input type="submit" id="btnAdd" class="btn
      btn-primary"
```

```
        value="Add"
        name="_eventId_customerInfoCollected" />
<button id="btnCancel" class="btn btn-
default" name="_eventId_cancel">Cancel
</button>
</div>
</div>

</fieldset>
</form:form>
</section>
</body>
</html>
```

2. Similarly, create one more JSP View file called `collectShippingDetail.jsp` under the same directory, add the following code snippet to it, and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset="utf-8">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Customer</title>
</head>
<body>
<section>
<div class="jumbotron">
<div class="container">
<h1>Shipping</h1>
<p>Shipping details</p>
</div>
</div>
</section>
<section class="container">
<form:form modelAttribute="order.shippingDetail"
class="form-horizontal">
<fieldset>
<legend>Shipping Details</legend>
<div class="form-group">
<label class="control-label col-lg-2 col-lg-2"
for="name" />Name</label>
```

```
<div class="col-lg-10">
  <form:input id="name" path="name"
    type="text" class="form:input-large" />
</div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2 col-lg-2"
    for="shippingDate" />shipping
    Date (dd/mm/yyyy)</label>
  <div class="col-lg-10">
    <form:input id="shippingDate"
      path="shippingDate" type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="doorNo">Door No</label>
  <div class="col-lg-10">
    <form:input id="doorNo"
      path="shippingAddress.doorNo" type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="streetName">Street Name</label>
  <div class="col-lg-10">
    <form:input id="streetName"
      path="shippingAddress.streetName."
      type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="areaName">Area Name</label>
  <div class="col-lg-10">
    <form:input id="areaName"
      path="shippingAddress.areaName"
      type="text"
      class="form:input-large" />
  </div>
</div>
```



```
<div class="form-group">
  <label class="control-label col-lg-2"
    for="state">State</label>
  <div class="col-lg-10">
    <form:input id="state"
      path="shippingAddress.state" type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="country">country</label>
  <div class="col-lg-10">
    <form:input id="country"
      path="shippingAddress.country" type="text"
      class="form:input-large" />
  </div>
</div>

<div class="form-group">
  <label class="control-label col-lg-2"
    for="zipCode">Zip Code</label>
  <div class="col-lg-10">
    <form:input id="zipCode"
      path="shippingAddress.zipCode" type="text"
      class="form:input-large" />
  </div>
</div>

<input type="hidden" name="_flowExecutionKey"
  value="{flowExecutionKey}"/>

<div class="form-group">
  <div class="col-lg-offset-2 col-lg-10">
    <button id="back" class="btn btn-default"
      name="_eventId_backToCollectCustomerInfo">
      back</button>
    <input type="submit" id="btnAdd" class="btn
      btn-primary"
      value="Add"
      name="_eventId_shippingDetailCollected"/>
    <button id="btnCancel" class="btn btn-
      default"
      name="_eventId_cancel">Cancel</button>
  </div>
</div>
```

```
        </fieldset>
    </form:form>
</section>
</body>
</html>
```

3. Create one more JSP View file called `orderConfirmation.jsp` to confirm the order by the user, under the same directory, then add the following code snippet to it and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring"
    uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="fmt"
    uri="http://java.sun.com/jsp/jstl/fmt"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset="utf-8">
<link rel="stylesheet"
    href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
    /bootstrap.min.css">
<title>Order Confirmation</title>
</head>

<body>

    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Order</h1>
                <p>Order Confirmation</p>
            </div>
        </div>
    </section>
    <div class="container">
        <div class="row">
            <form:form modelAttribute="order" class="form-horizontal">
                <input type="hidden" name="_flowExecutionKey"
                    value="${flowExecutionKey}" />

                <div
                    class="well col-xs-10 col-sm-10 col-md-6 col-
                    xs-offset-1 col-sm-offset-1 col-md-offset-3">
                    <div class="text-center">
```

```
        <h1>Receipt</h1>
    </div>
    <div class="row">
        <div class="col-xs-6 col-sm-6 col-md-6">
            <address>
                <strong>Shipping Address</strong> <br>
                ${order.shippingDetail.name}<br>
                ${order.shippingDetail.shippingAddress.doorNo},
                ${order.shippingDetail.shippingAddress.streetName}
                <br>
                ${order.shippingDetail.shippingAddress.areaName},
                ${order.shippingDetail.shippingAddress.state}
                <br>
                ${order.shippingDetail.shippingAddress.country},
                ${order.shippingDetail.shippingAddress.zipCode}
                <br>
            </address>
        </div>
        <div class="col-xs-6 col-sm-6 col-md-6 text-right">
            <p>
                <em>Shipping DateDate:
                <fmt:formatDate type="date"
value="${order.shippingDetail.shippingDate}" /></em>
                </p>
            </div>
        </div>
    <div class="row">
        <div class="col-xs-6 col-sm-6 col-md-6">
            <address>
                <strong>Billing Address</strong> <br>
                ${order.customer.name}<br>
                ${order.customer.billingAddress.doorNo},
                ${order.customer.billingAddress.streetName}
                <br>
                ${order.customer.billingAddress.areaName},
                ${order.customer.billingAddress.state}
                <br>
                ${order.customer.billingAddress.country},
                ${order.customer.billingAddress.zipCode}
                <br>
                <abbr>P:</abbr>
                ${order.customer.phoneNumber}
            </address>
        </div>
    </div>
    <div class="row">

        <table class="table table-hover">
```

```
<thead>
  <tr>
    <th>Product</th>
    <th>#</th>
    <th class="text-center">Price</th>
    <th class="text-center">Total</th>
  </tr>
</thead>
<tbody>
  <c:forEach var="cartItem"
    items="${order.cart.cartItems}">
    <tr>
      <td class="col-md-9">
        <em>${cartItem.product.name}</em></td>
      <td class="col-md-1"
        style="text-align: center">
        ${cartItem.quantity}</td>
      <td class="col-md-1 text-
        center">${cartItem.product.unitPrice}
      </td>
      <td class="col-md-1 text-
        center">${cartItem.totalPrice}
      </td>
    </tr>
  </c:forEach>

  <tr>
    <td> </td>
    <td> </td>
    <td class="text-right"><h4>
      <strong>Grand Total:
    </strong>
    </h4></td>
    <td class="text-center text-
      danger"><h4>
      <strong>${order.cart.grandTotal}
    </strong>
    </h4></td>
  </tr>
</tbody>
</table>
<button id="back" class="btn btn-default"
name="_eventId_backToCollectShippingDetail">
back</button>

<button type="submit" class="btn btn-
success"
  name="_eventId_orderConfirmed">
```

```
        Confirm    <span class="glyphicon glyphicon-chevron-right"></span>
    </button>
    <button id="btnCancel" class="btn btn-default"
        name="_eventId_cancel">Cancel</button>
    </div>
</div>
</form:form>
</div>
</div>
</body>
</html>
```

4. Next, we need to create another JSP View file called `invalidCartWarning.jsp` to show an error message in case the cart is empty at the checkout; add the following code snippet to `invalidCartWarning.jsp` and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset="utf-8">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Invalid cart </title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1 class="alert alert-danger"> Empty Cart</h1>
            </div>
        </div>
    </section>

    <section>
        <div class="container">
            <p>
                <a href="<spring:url value="/market/products" />"
                    class="btn btn-primary">
                    <span class="glyphicon-hand-left glyphicon">
</span> products
                </a>
            </p>
```

```
        </div>
    </section>
</body>
</html>
```

5. To thank the customer after a successful checkout flow, we need to create one more JSP View file, called `thankCustomer.jsp`, as follows:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
    uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset="utf-8">
<link rel="stylesheet"
    href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
    /bootstrap.min.css">
<title>Invalid cart </title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1 class="alert alert-danger"> Thank you</h1>
                <p>Thanks for the order. your order will be
                    delivered to you on
                <fmt:formatDate type="date"
                    value="${order.shippingDetail.shippingDate}" />
                </p>
                <p>Your Order Number is ${order.orderId}</p>
            </div>
        </div>
    </section>

    <section>
        <div class="container">
            <p>
                <a href="<spring:url value="/market/products" />"
                    class="btn btn-primary">
                    <span class="glyphicon-hand-left glyphicon">
                    </span> products
                </a>
            </p>
        </div>
    </section>
```

```
</body>
</html>
```

6. If the user cancels the checkout in any of the Views, we need to show the checkout cancelled message; for that we need to have a JSP file called `checkOutCancelled.jsp` as follows:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
    uri="http://www.springframework.org/tags"%>

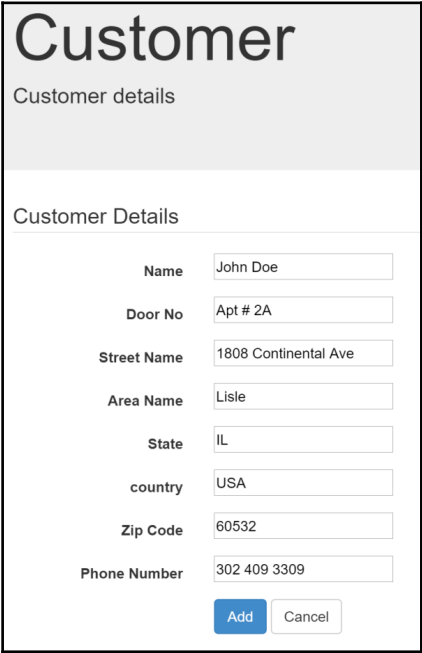
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset="utf-8">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Invalid cart </title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1 class="alert alert-danger">check out cancelled</h1>
                <p>Your Check out process cancelled! you may
                    continue shopping..</p>
            </div>
        </div>
    </section>

    <section>
        <div class="container">
            <p>
                <a href="<spring:url value="/market/products" />"
                    class="btn btn-primary">
                    <span class="glyphicon-hand-left glyphicon">
                    </span> products
                </a>
            </p>
        </div>
    </section>
</body>
</html>
```

7. As a last step, open `cart.jsp` from `src\main\webapp\WEB-INF\views\` and assign the value `<spring:url value="/checkout?cartId=${cartId}" />` to the `href` attribute of the checkout link, as follows:

```
<a href= "<spring:url value="/checkout?cartId=${cartId}"/>"
class="btn btn-success pull-right">
  <span class="glyphicon-shopping-cart glyphicon">
</span> Check out
</a>
```

8. With all these steps executed, now run the application and enter the `http://localhost:8080/webstore/market/products` URL. Next, click on the **Details** button of any of the products and click on the **Order Now** button from the product details page to add products to the shopping cart. Now go to the cart page by clicking the **View Cart** button; you will be able to see our **Checkout** button on that page. Just click on the **Checkout** button; you will be able to see a web page as follows to collect the customer info:



The screenshot shows a web form titled "Customer" with the subtitle "Customer details". Below this is a section labeled "Customer Details" containing several input fields for customer information. The fields are labeled "Name", "Door No", "Street Name", "Area Name", "State", "country", "Zip Code", and "Phone Number". Each field contains a sample value: "John Doe", "Apt # 2A", "1808 Continental Ave", "Lisle", "IL", "USA", "60532", and "302 409 3309" respectively. At the bottom of the form are two buttons: "Add" (highlighted in blue) and "Cancel".

Customer details collection form

9. After furnishing all the customer details, if you press the **Add** button, Spring Web Flow will take you to the next view state, which is to collect shipping details, and so on up to confirming the order. Upon confirming the order, Spring Web Flow will show you the thank you message View as the end state.

What just happened?

What we have done from steps 1 to 6 is a repeated task, creating JSP View files for each view state. We defined the `model` attribute for each view state in `checkout-flow.xml`:

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
  model="order">
  <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

That Model object gets bound to the View via the `modelAttribute` attribute of the `<form:form>` tag, as follows:

```
<form:form modelAttribute="order.customer" class="form-horizontal">
  <fieldset>
    <legend>Customer Details</legend>

    <div class="form-group">
      <label class="control-label col-lg-2"
for="name">Name</label>
      <div class="col-lg-10">
        <form:input id="name" path="name" type="text"
class="form-input-large" />
      </div>
    </div>
```

In the preceding snippet of `collectCustomerInfo.jsp`, you can see that we have bound the `<form:input>` tag to the `name` field of the `customer` object, which comes from the Model object (`order.customer`). Similarly, we have bound the `shippingDetail` and `order` objects to `collectShippingDetail.jsp` and `orderConfirmation.jsp` respectively.

It's good that we have bound the `Order`, `Customer`, and `ShippingDetail` objects to the Views, but what will happen after clicking the **submit** button in each View, or say, the **Cancel** or **back** buttons? To know the answer, we need to investigate the following code snippet from `collectCustomerInfo.jsp`:

```
<input type="submit" id="btnAdd" class="btn btn-primary" value="Add"
  name="_eventId_customerInfoCollected" />
```

On the surface, the preceding `<input>` tag just acts as a submit button, but the real difference comes from the `name` attribute (`name="_eventId_customerInfoCollected"`). We have assigned the value `_eventId_customerInfoCollected` to the `name` attribute of the `<input>` tag for a purpose. The purpose is to instruct Spring Web Flow to raise an event on submission of this form.

When this form is submitted, Spring Web Flow raises an event based on the `name` attribute. Since we have assigned a value with the `_eventId_` prefix (`_eventId_customerInfoCollected`), Spring Web Flow recognizes it as the event name and raises an event with the name `customerInfoCollected`.

As we already learned, transitions from one state to another state happen with the help of events, so on submitting the `collectCustomerInfo` form, Spring Web Flow takes us to the next view state, which is `collectShippingDetail`:

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
model="order">
    <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

Similarly, we raise events when clicking the **Cancel** or **back** buttons; see the following code snippet from `collectShippingDetail.jsp`:

```
<button id="back" class="btn btn-default"
name="_eventId_backToCollectCustomerInfo">back</button>

<button id="btnCancel" class="btn btn-default"
name="_eventId_cancel">Cancel</button>
```

Okay, so we understand how to raise Spring Web Flow events from a View to directly transition from one view state to another state. However, we need to understand one more important concept regarding Spring Web Flow execution—each flow execution is identified by the flow execution key at runtime. During flow execution, when a view state is entered, the flow execution pauses and waits for the user to perform some action (such as entering some data in the form). When the user submits the form or chooses to cancel the form, the flow execution key is sent along with the form data, in order for the flow to resume where it left off. We can do that with the help of the hidden `<input>` tag, as follows:

```
<input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}"/>
```

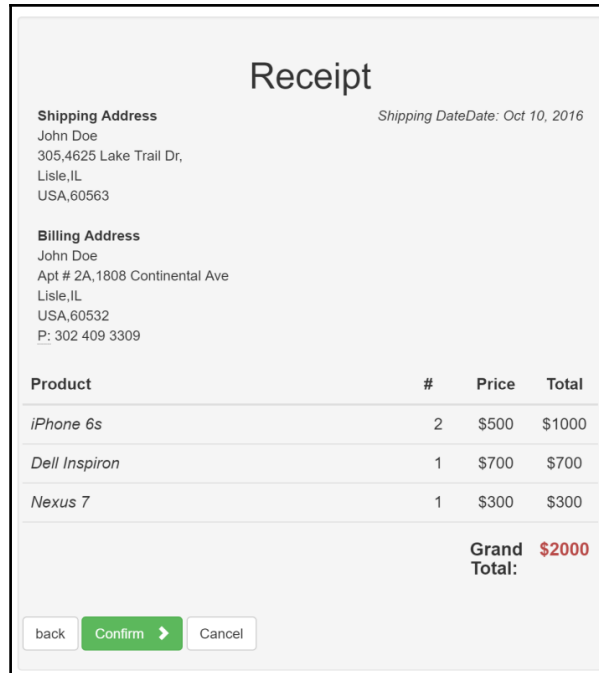
If you look carefully, we have this tag in every flow-related View file, such as `collectCustomerInfo.jsp`, `collectShippingDetail.jsp`, and so on. Spring Web Flow stores a unique flow execution key under the model attribute name `flowExecutionKey` in every flow-related View; we need to store this value in the form variable called `_flowExecutionKey` in order for it to be identified by Spring Web Flow.

So that's all about the View files associated with our checkout flow definition. But we need to invoke the flow upon clicking the **Checkout** button from the cart page. As we already learned to invoke our checkout flow, we need to fire a web request with the cart ID as the request parameter, so in step 7 we have changed the `href` attribute of the checkout link, to

form a request URL something similar to

`http://localhost:8080/webstore/checkout?cartId=55AD1472D4EC`.

So now, if you click on the **Checkout** button after selecting some products and placing them in the shopping cart, you will be able to initiate the checkout flow; the following screenshot shows the order confirmation page that will be shown as an outcome of reaching the `orderConfirmation` state:



The screenshot shows a receipt page with the title "Receipt" at the top center. Below the title, there are two address sections: "Shipping Address" and "Billing Address". The shipping address is for John Doe at 305,4625 Lake Trail Dr, Lisle, IL, USA, 60563. The billing address is for John Doe at Apt # 2A, 1808 Continental Ave, Lisle, IL, USA, 60532, with a phone number P: 302 409 3309. To the right of the shipping address, it says "Shipping Date: Oct 10, 2016". Below the addresses is a table with four columns: "Product", "#", "Price", and "Total". The table lists three items: "iPhone 6s" (2 units at \$500 each, total \$1000), "Dell Inspiron" (1 unit at \$700, total \$700), and "Nexus 7" (1 unit at \$300, total \$300). At the bottom right of the table, it says "Grand Total: \$2000". At the bottom of the page, there are three buttons: "back", "Confirm" (with a right arrow), and "Cancel".

Product	#	Price	Total
iPhone 6s	2	\$500	\$1000
Dell Inspiron	1	\$700	\$700
Nexus 7	1	\$300	\$300

Grand Total: \$2000

back Confirm > Cancel

Order confirmation view state

Have a go hero – adding a decision state

Although we have finished our checkout flow, there is still a bit of room to improve the flow. Every time the checkout flow starts, it collects the customer details, but what if a returning customer makes an order—they probably don't want to fill in their details each time. You can autofill returning customer details from existing records. You can also update the inventory of products upon confirmation.

Here are some of the improvements you can make to avoid collecting customer details for returning customers:

- Create a customer repository and service layer to store, retrieve, and find customer objects. You can probably have methods such as the following in your `CustomerRepository` and `CustomerService` interfaces, and in their corresponding implementation classes:
 - `public void saveCustomer(Customer customer)`
 - `public Customer getCustomer(String customerId)`
 - `public Boolean isCustomerExist(String customerId)`
- Define a view state in `checkout-flow.xml` to collect customer IDs. Don't forget to create the corresponding JSP View file to collect customer IDs.
- Define a decision state in `checkout-flow.xml` to check whether a customer exists in `CustomerRepository`, through `CustomerService`. Based on the returning Boolean value, direct the transition to collect the customer details view state or prefill the `order.customer` object from `CustomerRepository`. The following is the sample decision state:

```
<decision-stateid="checkCustomerExist">
<if test="customerServiceImpl.isCustomerExist(order.customer.
customerId) "
then=" collectShippingDetail"
else=" collectCustomerInfo"/>
</decision-state>
```

- After collecting customer details, don't forget to store them in `CustomerRepository` through an action state. Similarly, fill in the `order.customer` object after the decision state.

Summary

We only saw the very minimum of the concepts required to get a quick overview of the Spring Web Flow framework in this chapter. At the start of this chapter, we learned some of the basic concepts of the Spring Web Flow framework, and then we created the checkout flow for our web store application. We also learned how to fire a Web Flow event from a View. In the next chapter, we will learn more about how to incorporate the Apache Tiles framework into Spring MVC.

11

Template with Tiles

When it comes to web application development, reusability and maintenance are two important factors that need to be considered. Apache Tiles is another popular open source framework that encourages reusable template-based web application development.

In this chapter, you are going to see how to incorporate the Apache Tiles framework within a Spring MVC application, so that we can obtain maximum reusability of frontend templates with the help of Apache Tiles. Apache Tiles are mostly used to reduce redundant code in the frontend by leveraging frontend templates. After finishing this chapter, you will have a basic idea about decomposing pages using reusable Apache Tile templates.

Enhancing reusability through Apache Tiles

In the past, we developed a series of web pages (Views) as part of our `webstore` application, such as a page to show products, another page to add products, and so on. Although every View has served a different purpose, all of them share a common visual pattern; each page has a header, a content area, and so on. We hardcoded and repeated those common elements in every JSP View page. But this is not a good idea because, in future, if we want to change the look and feel of any of these common elements, we have to change every page in order to maintain a consistent look and feel across all the web pages.

To address this problem, modern web applications use template mechanisms; Apache Tiles is one such template composition framework. Tiles allow developers to define reusable page fragments (tiles), which can be assembled into a complete web page at runtime. These fragments can have parameters to allow dynamic content. This increases the reusability of templates and reduces code duplication.

Time for action – creating Views for every View state

Okay, enough introduction, let's dive into Apache Tiles by defining a common layout for our web application and let the pages extend the layout:

1. Open `pom.xml`. You can find `pom.xml` under the project root directory itself.
2. You will be able to see some tabs under `pom.xml`; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter `org.apache.tiles` as **Group Id**, `tiles-extras` as **Artifact Id**, `3.0.5` as **Version**, and select **Scope** as **compile**. Then click on the **OK** button and save `pom.xml`.
4. Now create a directory structure called `layouts/definitions/` under the `src/main/webapp/WEB-INF/` directory and create an XML file called `tiles.xml`. Add the following content to it and save it:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache
Software Foundation//DTD Tiles Configuration 3.0//EN"
"http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

<tiles-definitions>
    <definition name="baseLayout" template="/WEB-
INF/layouts/template/baseLayout.jsp">
        <put-attribute name="title" value="Sample
Title" />
        <put-attribute name="heading" value="" />
        <put-attribute name="tagline" value="" />
        <put-attribute name="navigation" value="/WEB-
INF/layouts/template/navigation.jsp" />
        <put-attribute name="content" value="" />
        <put-attribute name="footer" value="/WEB-
INF/layouts/template/footer.jsp" />
    </definition>
    <definition name="welcome" extends="baseLayout">
        <put-attribute name="title" value="Products" />
        <put-attribute name="heading" value="Products"
/>
        <put-attribute name="tagline" value="All the
available products in our store" />
        <put-attribute name="content" value="/WEB-
INF/views/products.jsp" />
    </definition>
    <definition name="products" extends="baseLayout">
        <put-attribute name="title" value="Products" />
        <put-attribute name="heading" value="Products"
```

```
</>
    <put-attribute name="tagline" value="All the
available products in our store" />
    <put-attribute name="content" value="/WEB-
INF/views/products.jsp" />
</definition>
<definition name="product" extends="baseLayout">
    <put-attribute name="title" value="Product" />
    <put-attribute name="heading" value="Product"
/>
    <put-attribute name="tagline" value="Details"
/>
    <put-attribute name="content" value="/WEB-
INF/views/product.jsp" />
</definition>
<definition name="addProduct"
extends="baseLayout">
    <put-attribute name="title" value="Products" />
    <put-attribute name="heading" value="Products"
/>
    <put-attribute name="tagline" value="Add
Product" />
    <put-attribute name="content" value="/WEB-
INF/views/addProduct.jsp" />
</definition>
<definition name="login" extends="baseLayout">
    <put-attribute name="title" value="Login" />
    <put-attribute name="heading" value="Welcome to
Web Store!" />
    <put-attribute name="tagline" value="The one
and only amazing web store" />
    <put-attribute name="content" value="/WEB-
INF/views/login.jsp" />
</definition>
<definition name="cart" extends="baseLayout">
    <put-attribute name="title" value="Shopping
Cart" />
    <put-attribute name="heading" value="Cart" />
    <put-attribute name="tagline" value="All the
selected products in your cart" />
    <put-attribute name="content" value="/WEB-
INF/views/cart.jsp" />
</definition>
</tiles-definitions>
```

5. Now create a directory called `template` under the `src/main/webapp/WEB-INF/layouts/` directory and create a JSP file called `baseLayout.jsp`. Add the following content to it and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="tiles"
uri="http://tiles.apache.org/tags-tiles"%>

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width,
initial-scale=1.0">

<title><tiles:insertAttribute name="title" /></title>

<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">

<script
src="https://ajax.googleapis.com/ajax/libs
/angularjs/1.5.1/angular.min.js"></script>

<script
src="/webstore/resources/js/controllers.js"></script>

</head>

<body>
  <section class="container">
    <div class="pull-right" style="padding-right:
50px">
      <a href="?language=en">English</a>|<a
href="?language=nl">Dutch</a>
      <a href="<c:url value="/logout"
/>">Logout</a>
    </div>
  </section>

  <div class="container">
```



```
<div class="jumbotron">
  <div class="header">
    <ul class="nav nav-pills pull-right">
      <tiles:insertAttribute
name="navigation" />
    </ul>
    <h3 class="text-muted">Web Store</h3>
  </div>

  <h1>
    <tiles:insertAttribute name="heading" />
  </h1>
  <p>
    <tiles:insertAttribute name="tagline" />
  </p>
</div>

<div class="row">
  <tiles:insertAttribute name="content" />
</div>

<div class="footer">
  <tiles:insertAttribute name="footer" />
</div>

</div>
</body>
</html>
```

6. Under the same directory (template), create another template JSP file called navigation.jsp and add the following content to it:

```
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<li><a href="<spring:url
value="/market/products"/>">Home</a></li>
<li><a href="<spring:url
value="/market/products/">">Products</a></li>
<li><a href="<spring:url
value="/market/products/add/">">Add Product</a></li>
<li><a href="<spring:url
value="/cart/">">Cart</a></li>
```

7. Similarly, create one last template JSP file called `footer.jsp` and add the following content to it:

```
<p>&copy; Company 2016</p>
```

8. Now we have created the common base layout template and the tile definition for all our pages, we need to remove the common page elements from all our JSP View files. For example, if you remove all the common elements, such as the jumbotron section and others, from our `products.jsp` file, and keep only the container section, it would look as follows:



Do not remove the tag lib references and link references.

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<section class="container">
  <div class="row">
    <c:forEach items="${products}" var="product">
      <div class="col-sm-6 col-md-3"
style="padding-bottom: 15px">
        <div class="thumbnail">
          </c:url>"
alt="image" style="width: 100%" />
          <div class="caption">
            <h3>${product.name}</h3>
            <p>${product.description}</p>
            <p>${product.unitPrice}</p>
            <p>Available
${product.unitsInStock} units in stock</p>
            <p>
              <a
                href=" <spring:url
value="/market/product?id=${product.productId}" /> "
                class="btn btn-primary">
<span class="glyphicon-info-sign
glyphicon" /></span> Details
              </a>
            </p>
          </div>
        </div>
      </c:forEach>
    </div>
  </div>
</section>
```

```
        </div>
    </div>
</div>
</c:forEach>
</div>
</section>
```

9. Similarly remove all the common elements other than the main container section from every JSP View file that is under the `/src/main/webapp/WEB-INF/views` directory; do not remove the `taglib` references.
10. Now create a Tiles configuration called `TilesConfig` under the `com.packt.webstore.config` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context
    .annotation.Configuration;
import org.springframework.web.servlet
    .view.UrlBasedViewResolver;
import org.springframework.web.servlet
    .view.tiles3.TilesConfigurer;
import org.springframework.web.servlet
    .view.tiles3.TilesView;

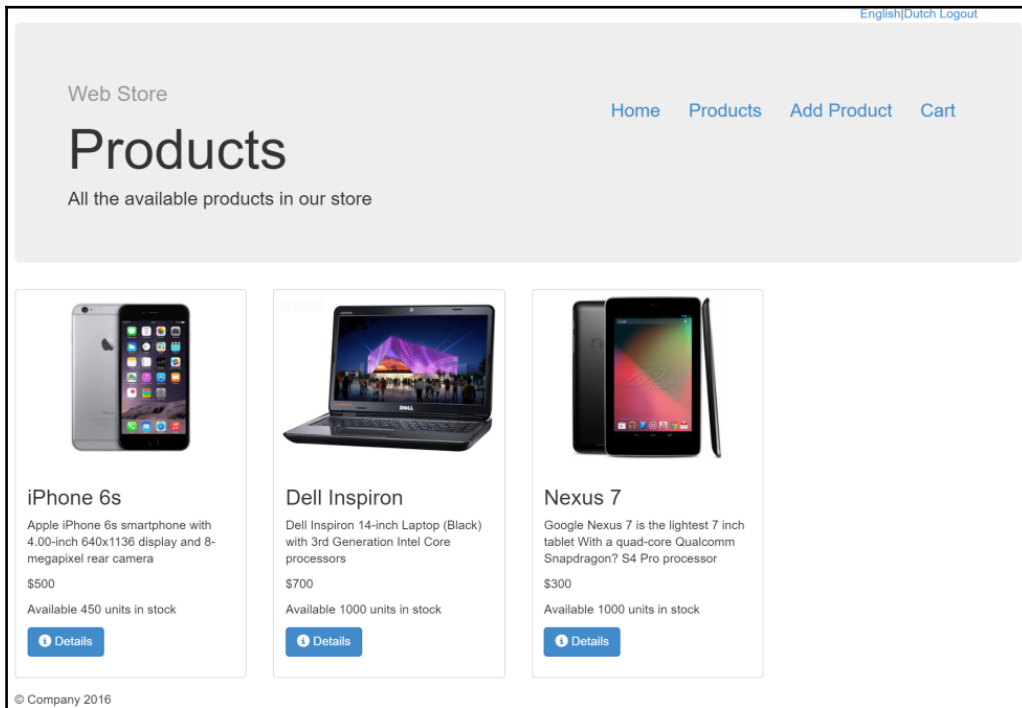
@Configuration
public class TilesConfig {

    @Bean
    public UrlBasedViewResolver viewResolver() {
        UrlBasedViewResolver viewResolver = new
        UrlBasedViewResolver();
        viewResolver.setViewClass
        (TilesView.class);
        viewResolver.setOrder(-2);
        return viewResolver;
    }

    @Bean
    public TilesConfigurer tilesConfigurer() {
        TilesConfigurer tilesConfigurer = new
        TilesConfigurer();
        tilesConfigurer.setDefinitions("/WEB-
        INF/layouts/definitions/tiles.xml");
        tilesConfigurer.setCheckRefresh(true);
    }
}
```

```
        return tilesConfigurer;  
    }  
}
```

11. Now run our application and enter the URL `http://localhost:8080/webstore/market/products`. You will be able to see our regular products page with an extra navigation bar at the top and a footer at the bottom. You can add a product by clicking on the **Add Product** link:



Products page with the Apache Tiles View

What just happened?

To work with Apache Tiles, we need Apache Tiles related JARs, so from steps 1 to 3 we added those JARs via Maven dependencies. Step 4 is very important because we created our tiles definition file (`tiles.xml`) in that step. Understanding the tiles definition file is crucial to developing Apache Tiles-based applications, so you need to understand our tile definition file.

A tile definition file is a collection of definitions, where each definition can be associated with a template via the `template` attribute for the layout:

```
<definition name="baseLayout" template="/WEB-INF/layouts/template/baseLayout.jsp">
  <put-attribute name="title" value="Sample Title" />
  <put-attribute name="heading" value="Sample Heading" />
  <put-attribute name="tagline" value="Sample Tagline" />
  <put-attribute name="navigation" value="/WEB-INF/layouts/template/navigation.jsp" />
  <put-attribute name="content" value="" />
  <put-attribute name="footer" value="/WEB-INF/layouts/template/footer.jsp" />
</definition>
```

Within each definition, we can define many attributes. These attributes can be a simple text value or a full-blown markup file. These attributes would be available in the template file via the `<tiles:insertAttribute>` tag. For example, if you open the base layout (`baseLayout.jsp`) template, you can see the following snippet under the `jumbotron<div>` tag:

```
<h1>
  <tiles:insertAttribute name="heading" />
</h1>
<p>
  <tiles:insertAttribute name="tagline" />
</p>
```

So at runtime, Apache Tiles would replace the `<tiles:insertAttribute name="heading" />` tag with the value `Sample Heading` and similarly the `<tiles:insertAttribute name="tagline" />` tag with the value `Sample Tagline`.

So the `baseLayout` definition is associated with the template `/WEB-INF/layouts/template/baseLayout.jsp`, and we can insert the defined attributes such as `tile`, `heading`, `tagline`, and more in the template using the `<tiles:insertAttribute>` tag.

Apache Tiles allows us to extend a definition just like how we extend a Java class, so that the defined attributes would be available for the derived definition, and we can even override those attributes values if we want. For example, look at the following definition from `tile-definition.xml`:

```
<definition name="products" extends="baseLayout">
  <put-attribute name="title" value="Products" />
  <put-attribute name="heading" value="Products" />
```

```
<put-attribute name="tagline" value=" All the available products in our
store" />
<put-attribute name="content" value="/WEB-INF/views/products.jsp" />
</definition>
```

This definition is an extension of the `baseLayout` definition. We have only overridden the `title`, `heading`, `tagline`, and `content` attributes, and since we have not defined any template for this definition, it uses the same template that we configured for the `baseLayout` definition.

Similarly, we defined the tile definition for every possible logical View name that can be returned from our Controllers. Note that each definition name (except the `baseLayout` definition) is a Spring MVC logical View name.

From steps 5 to 7, we just created the templates that can be used in the tile definition. First, we created the base layout template (`baseLayout.jsp`), then the navigation template (`navigation.jsp`), and finally the footer template (`footer.jsp`).

Steps 8 and 9 explained how to remove the existing redundant content such as the `jumbotron<div>` tag from every JSP View page. Note you have to be careful while doing this—don't accidentally remove the `taglib` references.

In step 10, we defined our `UrlBasedViewResolver` for `TilesView` in order to resolve logical View names into the tiles View and also configured the `TilesConfigurer` to locate the tiles definition files by the Apache Tiles framework.

That's it; if you run our application and enter the URL `http://localhost:8080/webstore/products`, you will be able to see our regular products page with an extra navigation bar at the top and a footer at the bottom, as mentioned in step 11. You can add a product page by clicking on the **Add Product** link. Previously, every time a logical View name was returned by the Controller method, the `InternalResourceViewResolver` comes into action and finds the corresponding `jsp` View for the given logical View name. Now for every logical View name, the `UrlBasedViewResolver` will come into action and compose the corresponding View based on the template definition.

Pop quiz – Apache Tiles

Which of the following statements are true according to Apache Tiles?

1. The logical View name returned by the Controller must be equal to the `<definition>` tag name.
2. The `<tiles:insertAttribute>` tag acts as a placeholder in the template.
3. A `<definition>` tag can extend another `<definition>` tag.
4. All of the above.

Summary

Apache Tiles is a separate framework. We only provided the bare minimum required concepts to get a quick overview of Apache Tiles. You saw how to use and leverage the Apache Tiles framework in order to ensure maximum reusability in the View files and maintain a consistent look and feel throughout all the web pages of our application.

In the next chapter, you will see how to test our web application using the various APIs provided by Spring MVC.

12

Testing Your Application

*For a web application developer, testing web applications is always a challenging task because getting a real-time test environment for web applications requires a lot of effort. But thanks to the **Spring MVC Test framework**, it simplifies the testing of Spring MVC applications.*

But why do we need to consider putting effort into testing our application? Writing good test cases for our application is a kind of like buying an insurance policy for your application, although it does not add any functional values to your application, it will definitely save you time and effort by detecting functionality failures early. Consider your application growing bigger and bigger in terms of functionality—you need some mechanism to ensure that the existing functionalities were not disturbed by means of introducing new functionalities.

Testing frameworks provide you with such a mechanism, to ensure that your application's behavior is not altered due to refactoring or the addition of new code. They also ensure existing functionalities work as expected.

In this chapter, we are going to look at the following topics:

- Testing the domain object and validator
- Testing Controllers
- Testing RESTful web services

Unit testing

In software development, unit testing is a software testing method in which the smallest testable parts of source code, called units, are individually and independently tested to determine whether they behave exactly as we expect. To unit test our source code, all we need is a test program that can run a bit of our source code (unit), provide some inputs to each unit, and check the results for the expected output. Most unit tests are written using some sort of test framework set of library code designed to make writing and running tests easier. One such framework is called **JUnit**. It is a unit testing framework for the Java programming language.

Time for action – unit testing domain objects

Let's see how to test one of our domain objects using the JUnit framework to ensure it functions as expected. In an earlier chapter, we created a domain object to represent an item in a shopping cart called `CartItem`. The `CartItem` class has a method called `getTotalPrice` to return the total price of that particular cart item, based on the product and number of items it represents. Let's test whether the `getTotalPrice` method behaves properly:

1. Open `pom.xml`, which you can find under the root directory of the project itself.
2. You will be able to see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter `junit` as **Group Id**, `junit` as **Artifact Id**, `4.12` as **Version**, select **Scope** as **test**, click on the **OK** button, and save `pom.xml`.
4. Now create a class called `CartItemTest` under the `com.packt.webstore.domain` package in the `src/test/java` source folder, add the following code to it, and save the file:

```
package com.packt.webstore.domain;
import java.math.BigDecimal;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class CartItemTest {

    private CartItem cartItem;

    @Before
```

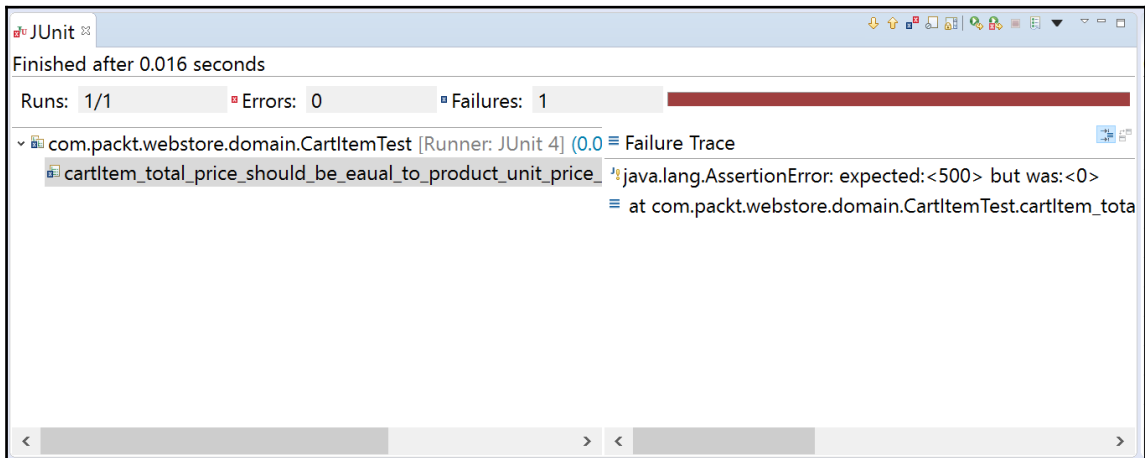
```
public void setup() {
    cartItem = new CartItem("1");
}

@Test
public void cartItem_total_price_should_be_equal_
to_product_unit_price_in_case_of_single_quantity() {
    //Arrange
    Product iphone = new Product("P1234", "iPhone
5s", new BigDecimal(500));
    cartItem.setProduct(iphone);

    //Act
    BigDecimal totalPrice =
cartItem.getTotalPrice();

    //Assert
    Assert.assertEquals(iphone.getUnitPrice(),
totalPrice);
}
```

5. Now right-click on `CartItemTest.java` and choose **Run As | JUnit Test**. You will see a failing test case in the **JUnit** window, as shown in the following screenshot:

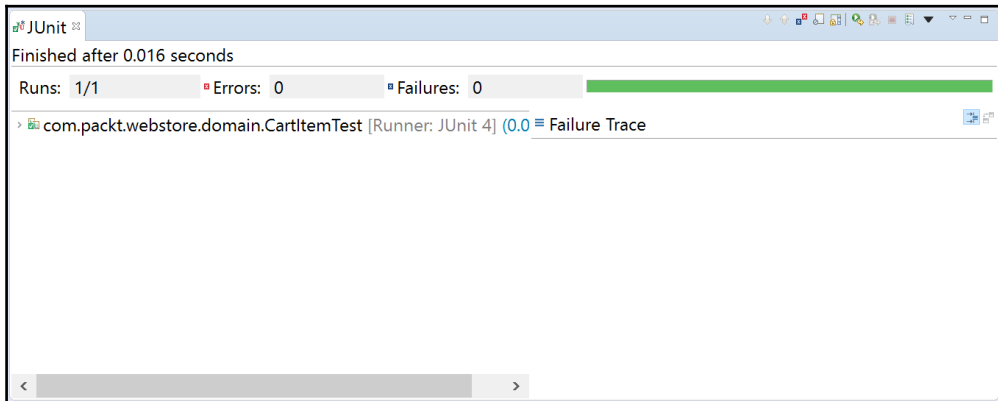


JUnit failing test case in CartItemTest

6. To make the test case pass, assign the value 1 to the `quantity` field of `CartItem` in the `setProduct` method of the `CartItem` class as follows, and save the file:

```
public void setProduct(Product product) {  
    this.product = product;  
    this.quantity = 1;  
    this.updateTotalPrice();  
}
```

7. Now right-click on `CartItemTest.java` again and choose **Run As | JUnit Test**. You will see a passing test case in the **JUnit** window, as shown in the following screenshot:



JUnit passing test case in `CartItemTest`

What just happened?

As I already mentioned, the `getTotalPrice` method of the `CartItem` class is designed to return the correct total price based on the product and the number of products it represents. But to ensure its behavior, we wrote a test program called `CartItemTest` under the `com.packt.webstore.domain` package in the `src/test/java` source folder, as mentioned in step 4.

In `CartItemTest`, we used some of the JUnit framework APIs such as the `@Test` and `@Before` annotations, and more. So in order to use these annotations in our `CartItemTest` class, prior to that we need to add the JUnit JAR as a dependency in our project. That's what we did from steps 1 to 3.

Now you need to understand the `CartItemTest` class thoroughly. The important method in the `CartItemTest` class is the `@Test` annotated method called `cartItem_total_price_should_be_equal_to_product_unit_price_in_case_of_single_quantity`. The `@Test` (`org.junit.Test`) annotation marks a particular method as a test method so that the JUnit framework can treat that method as a test method and execute it when we choose the **Run As | JUnit Test** option:

```
@Test
public void
cartItem_total_price_should_be_equal_to_product_unit_price_in_case_of_singl
e_quantity() {
    //Arrange
    Product iphone = new Product("P1234","iPhone 5s", new
BigDecimal(500));
    cartItem.setProduct(iphone);
    //Act
    BigDecimal totalPrice = cartItem.getTotalPrice();
    //Assert
    Assert.assertEquals(iphone.getUnitPrice(), totalPrice);
}
```

If you notice, this method was divided into three logical parts called Arrange, Act, and Assert:

- Arrange all the necessary preconditions and inputs to perform a test
- Act on the object or method under test
- Assert that the expected results have occurred

In the Arrange part, we just instantiated a product domain object (`iphone`) with a unit price value of 500 and added that product object to the `cartItem` object by calling `cartItem.setProduct(iphone)`. Now we have added a single product to the `cartItem`, but we haven't altered the quantity of the `cartItem` object. So if we call the `getTotalPrice` method of `cartItem`, we must get 500 (in `BigDecimal`), because the unit price of the domain object (`iphone`) we added in `cartItem` is 500.

In the Act part, we just called the method under test, which is the `getTotalPrice` method of the `cartItem` object, and stored the result in a `BigDecimal` variable called `totalPrice`. Later, in the Assert part, we used the JUnit API (`Assert.assertEquals`) to assert the equality between the `unitPrice` of the product domain object and the calculated `totalPrice` of `cartItem`:

```
Assert.assertEquals(iphone.getUnitPrice(), totalPrice);
```

The `totalPrice` of `cartItem` must be equal to the `unitPrice` of the product domain object, which we added to `cartItem`, because we added a single product domain object whose `unitPrice` needs to be the same as the `totalPrice` of `cartItem`.

When we run our `CartItemTest` as mentioned in step 5, the JUnit framework tries to execute all the `@Test` annotated methods in the `CartItemTest` class. So based on the Assertion result, a test case may fail or pass. In our case, our test case failed. You can see the failure trace showing an error message saying **expected <500> but was: <0>** in the screenshot for step 5. This is because in the Arrange part, when we added a product domain object to `cartItem`, it doesn't update the `quantity` field of the `cartItem` object. It is a bug, so to fix this bug we default the `quantity` field value to 1 whenever we set the product argument using the `setCartItem` method, as mentioned in step 6. Now finally, if we run our test case again, this time it passes as expected.

Have a go hero – adding tests for Cart

It's good that we tested and verified the `getTotalPrice` method of the `CartItem` class, but can you similarly write a test class for the `Cart` domain object class? In the `Cart` domain object class, there is method to get the grand total (`getGrandTotal`), and write various test case to check whether the `getGrandTotal` method works as expected.

Integration testing with the Spring Test context framework

When individual program units are combined and tested as a group, then it is known as **integration testing**. The Spring Test context framework provides first-class support for an integration test of a Spring-based application. We have defined lots of Spring managed beans in our web application context, such as services, repositories, view resolvers, and more, to run our application.

These managed beans are instantiated during the startup of an application by the Spring framework. While doing the integration testing, our test environment must also have those beans to test our application successfully. The Spring Test context framework gives us the ability to define a test context that is similar to the web application context. Let's see how to incorporate the Spring Test context to test our `ProductValidator` class.

Time for action – testing product validator

Let's see how we can boot up our test context using the Spring Test context framework to test our `ProductValidator` class:

1. Open `pom.xml`, which you can find under the root directory of the project itself.
2. You will be able to see some tabs at the bottom of the `pom.xml` file; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter `org.springframework` as **Group Id**, `spring-test` as **Artifact Id**, `4.3.0.RELEASE` as **Version**, and select **test** as **Scope**, then click on the **OK** button and save `pom.xml`.
4. Similarly, add one more dependency for **jsp-api**; repeat the same step with **Group Id** as `javax.servlet`, **Artifact Id** as `jsp-api`, and **Version** as `2.0`, but this time, select **Scope** as **test** and then click on the **OK** button and save `pom.xml`.
5. Next create a class called `ProductValidatorTest` under the `com.packt.webstore.validator` package in the `src/test/java` source folder, and add the following code to it:

```
package com.packt.webstore.validator;

import java.math.BigDecimal;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory
    .annotation.Autowired;
import org.springframework.test
    .context.ContextConfiguration;
import org.springframework.test.context
    .junit4.SpringJUnit4ClassRunner;
import org.springframework.test
    .context.web.WebAppConfiguration;
import org.springframework.validation.BindException;
import org.springframework
    .validation.ValidationUtils;

import com.packt.webstore.config
    .WebApplicationContextConfig;
import com.packt.webstore.domain.Product;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
    WebApplicationContextConfig.class)
```

```
@WebAppConfiguration
public class ProductValidatorTest {
    @Autowired
    private ProductValidator productValidator;
    @Test
    public void
product_without_UnitPrice_should_be_invalid() {
        //Arrange
        Product product = new Product();
        BindException bindException = new
BindException(product, " product");

        //Act
        ValidationUtils.invokeValidator
(productValidator, product, bindException);
        //Assert
        Assert.assertEquals(1,
bindException.getErrorCount());
        Assert.assertTrue(bindException
.getMessage().contains("Unit price is
Invalid. It cannot be empty."));
    }
    @Test
    public void
product_with_existing_productId_invalid() {
        //Arrange
        Product product = new Product("P1234","iPhone
5s", new BigDecimal(500));
        product.setCategory("Tablet");
        BindException bindException = new
BindException(product, " product");

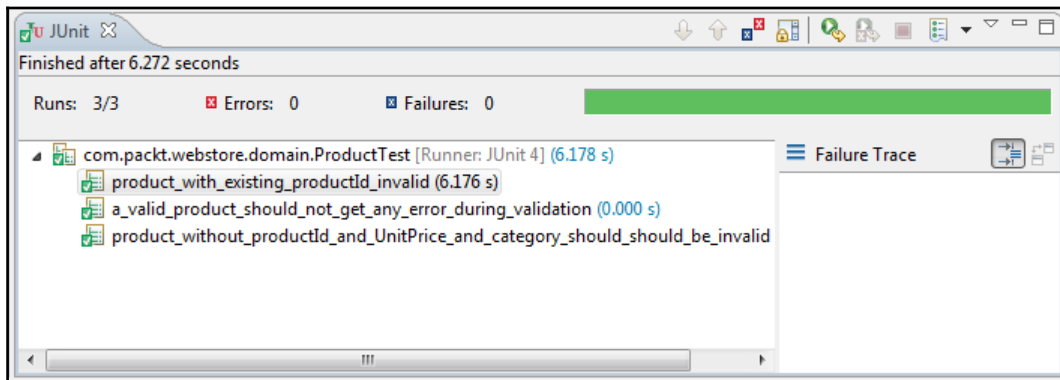
        //Act
        ValidationUtils.invokeValidator
(productValidator, product, bindException);
        //Assert
        Assert.assertEquals(1,
bindException.getErrorCount());
        Assert.assertTrue(bindException.
getMessage().contains("A product already
exists with this product id."));
    }
    @Test
    public void a_valid_product_should_not_get
_any_error_during_validation() {
        //Arrange
        Product product = new Product("P9876","iPhone
5s", new BigDecimal(500));
```

```
        product.setCategory("Tablet");
        BindException bindException = new
        BindException(product, " product");

        //Act
        ValidationUtils.invokeValidator
        (productValidator, product, bindException);
        //Assert
        Assert.assertEquals(0,
        bindException.getErrorCount());
    }

}
```

6. Now right-click on `ProductValidatorTest` and choose **Run As | JUnit Test**. You will be able to see passing test cases, as shown in the following screenshot:



Customer details collection form

What just happened?

As I already mentioned, Spring provides extensive support for integration testing. In order to develop a test case using the Spring Test context framework, we need the required `spring-test` JAR. In step 3, we just added a dependency to the `spring-test` JAR. The Spring Test context framework cannot run without the support of the `JUnit` JAR.

Step 5 is very important because it represents the actual test class (`ProductValidatorTest`) to test the validity of our `Product` domain object. The goal of the test class is to check whether all the validations (including bean validation and Spring validation) that were specified in the `Product` domain class are working. I hope you remember that we specified some of the bean validation annotations such as `@NotNull`, `@Pattern`, and more in the `Product` domain class.

One way to test whether those validations are taking place is by manually running our application and trying to enter invalid values. This approach is called manual testing. This is a very difficult job, whereas in automated testing we can write some test classes to run test cases in repeated fashion to test our functionality. Using JUnit, we can write this kind of test class.

The `ProductValidatorTest` class contains three test methods in total; we can identify a test method using the `@Test` (`org.junit.Test`) annotation of JUnit. Every test method can be logically separated into three parts, that is, *Arrange*, *Act*, and *Assert*. In the *Arrange* part, we instantiated and instrumented the required objects for testing; in the *Act* part, we invoked the actual functionality that needs to be tested; and finally in the *Assert* part, we compared the expected result and the actual result that is an output of the invoked functionality:

```
@Test
public void product_without_UnitPrice_should_be_invalid() {
    //Arrange
    Product product = new Product();
    BindException bindException = new BindException(product, "
product");

    //Act
    ValidationUtils.invokeValidator(productValidator, product,
bindException);
    //Assert
    Assert.assertEquals(1, bindException.getErrorCount());
    Assert.assertTrue(bindException.getLocalizedMessage().contains("Unit price
is Invalid. It cannot be empty."));
}
```

In the *Arrange* part of this test method, we just instantiated a bare minimum `Product` domain object. We have not set any values for the `productId`, `unitPrice`, and `category` fields. We purposely set up such a bare minimum domain object in the *Arrange* part to check whether our `ProductValidator` class is working properly in the *Act* part.

According to the `ProductValidator` class logic, the present state of the `product` domain object is invalid. And in the *Act* part, we invoked the `productValidator` method using

the `ValidationUtils` class to check whether the validation works or not. During validation, `productValidator` will store the errors in a `BindException` object. In the Arrange part, we simply check whether the `bindException` object contains one error using the JUnit Assert APIs, and check that the error message was as expected.

Another important thing you need to understand in our `ProductValidatorTest` class is that we used a Spring standard `@Autowired` annotation to get the instance of `ProductValidator`. The question here is who instantiated the `productValidator` object? The answer is in the `@ContextConfiguration` annotation. Yes, if you looked at the `classes` attribute specified in the `@ContextConfiguration` annotation, it has the name of our test context file (`WebApplicationContextConfig.class`).

If you remember correctly, you learned in the past that during the booting up of our application, Spring MVC creates a web application context (Spring container) with the necessary beans, as defined in the web application context configuration file. We need a similar kind of context even before running our test classes, so that we can use those defined beans (objects) in our test class to test them properly. The Spring Test framework makes this possible via the `@ContextConfiguration` annotation.

Likewise, we need a similar running application environment with all the resource files, and to achieve this we used the `@WebAppConfiguration` annotation from the Spring Test framework. The `@WebAppConfiguration` annotation instructs the Spring Test framework to load the application context as `WebApplicationContext`.

Now you have seen almost all the important things related to executing a Spring integration test, but there is one final configuration you need to understand, how to integrate JUnit and the Spring Test context framework into our test class. The `@RunWith(SpringJUnit4ClassRunner.class)` annotation just does this job.

So finally, when we run our test cases, we are able to see a green bar in the JUnit window indicating that the tests were successful.

Time for action – testing product Controllers

Now let's look at how to test our Controllers:

1. Create a class called `ProductControllerTest` under the `com.packt.webstore.controller` package in the `src/test/java` source folder and add the following code to it:

```
package com.packt.webstore.controller;
```

```
import static org.springframework.test.web.servlet
.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet
.result.MockMvcResultMatchers.model;

import java.math.BigDecimal;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory
.annotation.Autowired;
import org.springframework.test.context
.ContextConfiguration;
import org.springframework.test.context
.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context
.web.WebAppConfiguration;
import org.springframework.test.web
.servlet.MockMvc;
import org.springframework.test.web
.servlet.setup.MockMvcBuilders;
import org.springframework.web
.context.WebApplicationContext;

import com.packt.webstore.config
.WebApplicationContextConfig;
import com.packt.webstore.domain.Product;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
WebApplicationContextConfig.class)
@WebAppConfiguration
public class ProductControllerTest {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
    }

    @Test
    public void testGetProducts() throws Exception {
```

```
        this.mockMvc.perform(get("/market/products"))
            .andExpect(model().attributeExists("products"));
    }
    @Test
    public void testGetProductById() throws Exception
    {
        //Arrange
        Product product = new Product("P1234", "iPhone 5s",
            new BigDecimal(500));
        //Act & Assert
        this.mockMvc.perform(get("/market/product")
            .param("id", "P1234"))
            .andExpect(model().attributeExists("product"))
            .andExpect(model().attribute("product", product));
    }
}
```

2. Now right-click on the `ProductControllerTest` class and choose **Run As | JUnit Test**. You will be able to see that the test cases are being executed, and you will be able to see the test results in the **JUnit** window.

What just happened?

Similar to the `ProductValidatorTest` class, we need to boot up the test context and want to run our `ProductControllerTest` class as a Spring integration test. So we used similar annotations on top of `ProductControllerTest` as follows:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = WebApplicationContextConfig.class)
@WebAppConfiguration
public class ProductControllerTest {
```

As well as the two test methods that are available under `ProductControllerTest`, a single setup method is available as follows:

```
@Before
public void setup() {
    this.mockMvc =
    MockMvcBuilders.webAppContextSetup(this.wac).build();
}
```

The `@Before` annotation that is present on top the previous method indicates that this method should be executed before every test method. And within that method, we simply build our `mockMvc` object in order to use it in the following test methods. The `MockMvc` class is a special class provided by the Spring Test context framework to simulate browser actions within a test case, such as firing HTTP requests:

```
@Test
public void testGetProducts() throws Exception {
    this.mockMvc.perform(get("/market/products"))
        .andExpect(model().attributeExists("products"));
}
```

This test method simply fires a GET HTTP request to our application using the `mockMvc` object, and as a result, we ensure the returned model contains an attribute named `products`. Remember that our `list` method from the `ProductController` class is the thing handling the previous web request, so it will fill the model with the available products under the attribute name `products`.

After running our test case, you are able to see the green bar in the **JUnit** window, which indicates that the tests passed.

Time for action – testing REST Controllers

Similarly, we can test the REST-based Controllers as well—just follow these steps:

1. Open `pom.xml`, which you can find `pom.xml` under the root directory of the project itself.
2. You will be able to see some tabs at the bottom of `pom.xml` file; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; for **Group Id** enter `com.jayway.jsonpath`, for **Artifact Id** enter `json-path-assert`, for **Version** enter `2.2.0`, and for **Scope** select **test**, then click on the **OK** button and save `pom.xml`.
4. Now create a class called `CartRestControllerTest` under the `com.packt.webstore.controller` package in the `src/test/java` source folder, and add the following code to it:

```
package com.packt.webstore.controller;

import static org.springframework.test.web
    .servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web
```

```
.servlet.request.MockMvcRequestBuilders.put;
import static org.springframework.test.web
.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web
.servlet.result.MockMvcResultMatchers.status;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory
.annotation.Autowired;
import org.springframework.mock
.web.MockHttpSession;
import org.springframework.test.context
.ContextConfiguration;
import org.springframework.test
.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test
.context.web.WebAppConfiguration;
import org.springframework
.test.web.servlet.MockMvc;
import org.springframework.test.web
.servlet.setup.MockMvcBuilders;
import org.springframework.web.context
.WebApplicationContext;

import com.packt.webstore.config
.WebApplicationContextConfig;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
WebApplicationContextConfig.class)
@WebAppConfiguration
public class CartRestControllerTest {
    @Autowired
    private WebApplicationContext wac;

    @Autowired
    MockHttpSession session;
    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
    }
    @Test
    public void
read_method_should_return_correct_cart_json_object ()
```

```
throws Exception {
    //Arrange
    this.mockMvc.perform(put("/rest/cart/add/P1234")
        .session(session))
        .andExpect(status().is(200));

    //Act
    this.mockMvc.perform(get("/rest/cart/"+
        session.getId()).session(session))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.cartItems[0].
            product.productId").value("P1234"));
}

}
```

5. Now right-click on `CartRestControllerTest` and choose **Run As | JUnit Test**. You will be able to see that the test cases are being executed, and you will be able to see the test results in the **JUnit** window.

What just happened?

While testing REST Controllers, we need to ensure that the web response for the given web request contains the expected JSON object. To verify that, we need some specialized APIs to check the format of the JSON object. The `json-path-assert` JAR provides such APIs. We added a Maven dependency to the `json-path-assert` JAR from steps 1 to 3.

In step 4, we created our `CartRestControllerTest` to verify that our `CartRestController` class works properly. The `CartRestControllerTest` class is very similar to `ProductControllerTest`—the only difference is the way we assert the result of a web request. In `CartRestControllerTest`, we have one test method to test the `read` method of the `CartRestController` class.

The `read` method of `CartRestController` is designed to return a `cart` object as a JSON object for the given `cart ID`. In `CartRestControllerTest`, we tested this behavior in the `read_method_should_return_correct_cart_json_object` test method:

```
@Test
public void read_method_should_return_correct_cart_json_object ()
throws Exception {
    //Arrange
    this.mockMvc.perform(put("/rest/cart/add/P1234").session(session))
        .andExpect(status().is(200));

    //Act
    this.mockMvc.perform(get("/rest/cart/"+
```

```
session.getId()).session(session))
    .andExpect(status().isOk())
    .andExpect(jsonPath("$.cartItems[0].product.productId").value("P1234"));
}
```

In order to get a cart object for the given cart ID, we need to store the cart object in our cart repository first, through a web request. That is what we did in the Arrange part of the previous test method. The first web request we fired in the Arrange part adds a product domain object in the cart, whose ID is the same as the session ID.

In the Act part of the test case, we simply fired another REST based web request to get the cart object as JSON object. Remember we used the session ID as our cart ID to store our cart object, so while retrieving it, we need to provide the same session ID in the request URL. For this, we can use the mock session object given by the Spring Test framework. You can see that we auto-wired the session object in our `CartRestControllerTest` class:

```
this.mockMvc.perform(get("/rest/cart/"+ session.getId()).session(session))
    .andExpect(status().isOk())
    .andExpect (
        jsonPath("$.cartItems[0].product.productId").value("P1234"));
```

After we get the cart domain object as the JSON object, we have to verify whether it contains the correct product. We can do that with the help of the `jsonPath` method of `MockMvcResultMatchers`, as specified in the previous code snippets. After sending the REST web request to go get the cart object, we verified that the response status is okay and we also verified that the JSON object contains a product with the ID `P1234`.

Finally, when we run this test case, you can see the test cases being executed, and you can see the test results in the **JUnit** window.

Have a go hero – adding tests for the remaining REST methods

It's good that we tested and verified the `read` method of `CartRestController`, but we have not tested the other methods of `CartRestController`. You can add tests for the other methods of `CartRestController` in the `CartRestControllerTest` class to get more familiar with the Spring Test framework.

Summary

In this final chapter, you saw the importance of testing a web application and you saw how to test the validator using the Spring Test framework. You also found out how to test a normal Controller using the Spring Test context framework. As a last exercise, you also discovered how to test REST-based Controllers and how to use the mock session object from the Spring Test framework. During that exercise, you also learned how to use the JSON path library for assertion.

Thank you readers!

I am so thankful to all of you readers. I hope this book has helped you to understand Spring MVC from a beginner's perspective. I'm grateful for our connection. I hope all of you will continue to reach out to me at any time with feedback or questions. Thank you once again and all the very best in your exploration of the latest technologies!

Happy coding!



Using the Gradle Build Tool

*Throughout this book, we have used Apache Maven as our build tool, but there are other popular build tools also used widely in the Java community. One such build tool is Gradle. Instead of XML, Gradle uses a Groovy-based **Domain Specific Language (DSL)** as the base for the build script, which provides more flexibility when defining complex build scripts. Compared to Maven, Gradle takes less time for incremental builds. So, Gradle builds are very fast and effective for large projects.*

In this appendix, we will see how to install and use Gradle as the build tool in our project.

Installing Gradle

Perform the following steps to install Gradle:

1. Go to the Gradle download page by entering the URL `http://www.gradle.org/downloads` in your browser.
2. Click on the latest Gradle stable release download link; at the time of writing this, the stable release is `gradle-2.14.1`.
3. Once the download is finished, go to the downloaded directory and extract the ZIP file into a convenient directory of your choice.
4. Create an environment variable called `GRADLE_HOME`. Enter the extracted Gradle ZIP directory path as the value for the `GRADLE_HOME` environment variable.
5. Finally, append the `GRADLE_HOME` variable to `PATH` by simply appending the text; `%GRADLE_HOME%\bin` to the `PATH` variable.

Now that you have installed Gradle on your Windows-based computer, to verify whether the installation was completed correctly, go to the command prompt, type `gradle -v`, and press *Enter*. The output shows the Gradle version and also the local environment configuration.

The Gradle build script for your project

To configure the Gradle build script for your project, perform the following steps:

1. Go to the root directory of your project from the filesystem, create a file called `build.gradle`, and add the following content into the file and save it:

```
apply plugin: 'war'
apply plugin: 'eclipse-wtp'
repositories {
    mavenCentral() //add central maven repo to your buildfile
}

dependencies {

    compile 'org.springframework:spring-webmvc:4.3.0.RELEASE',

        'javax.servlet:jstl:1.2',
        'org.springframework:spring-jdbc:4.3.0.RELEASE',
        'org.hsqldb:hsqldb:2.3.2',
        'commons-fileupload:commons-fileupload:1.2.2',
        'org.apache.commons:commons-io:1.3.2',
        'org.springframework:spring-oxm:4.3.0.RELEASE',
        'org.codehaus.jackson:jackson-mapper-asl:1.9.10',
        'com.fasterxml.jackson.core:jackson-databind:2.8.0',
        'log4j:log4j:1.2.17',
        'org.springframework.security:spring-security
        -config:4.1.1.RELEASE',
        'org.springframework.security:spring-security
        -web:4.1.1.RELEASE',
        'org.hibernate:hibernate-validator:5.2.4.Final',
        'org.springframework.webflow:spring
        -webflow:2.4.2.RELEASE',
        'org.apache.tiles:tiles-extras:3.0.5'

    providedCompile 'javax.servlet:javax.servlet-api:3.1.0'

    testCompile 'junit:junit:4.12',
        'org.springframework:spring
        -test:4.3.0.RELEASE',
```

```
        'javax.servlet:jsp-api:2.0',  
        'com.jayway.jsonpath:json-path  
        -assert:2.2.0'  
    }  
}
```

2. Now go to the root directory of your project from the command prompt and issue the following command:

```
> gradle eclipse
```

3. Next, open a new workspace in your STS, go to **File | Import**, select the **Existing Projects into Workspace** option from the tree list (you can find this option under the **General** node), and then click on the **Next** button.
4. Click on the **Browse** button to select the root directory and locate your project directory. Click on **OK** and then on **Finish**.

Now, you will be able to see your project configured with the right dependencies in your STS.

Understanding the Gradle script

A task in Gradle is similar to a goal in Maven. The Gradle script supports many in-built plugins to execute build-related tasks. One such plugin is the `war` plugin, which provides many convenient tasks to help you build a web project. We can incorporate these tasks in our build script easily by applying a plugin in our Gradle script as follows:

```
apply plugin: 'war'
```

Similar to the `war` plugin, there is another plugin called `eclipse-wtp` to incorporate tasks related to converting a project into an eclipse project. The `eclipse` command we used in step 2 is actually provided by the `eclipse-wtp` plugin.

Inside the `repositories` section, we can define our remote binary repository location. When we build our Gradle project, we use this remote binary repository to download the required JARs. In our case, we defined our remote repository as the Maven central repository, as follows:

```
repositories {  
    mavenCentral()  
}
```

All of the project dependencies need to be defined inside of the `dependencies` section grouped under the scope declaration, such as `compile`, `providedCompile`, and `testCompile`. Consider the following code snippet:

```
dependencies {  
    compile  
    'org.springframework:spring-webmvc:4.3.0.RELEASE',  
    'javax.servlet:jstl:1.2'.  
}
```

If you look closely at the following dependency declaration line, the `compile` scope declaration, you see that each dependency declaration line is delimited with a `:` (colon) symbol, as follows:

```
'org.springframework:spring-webmvc:4.3.0.RELEASE'
```

The first part of the previous line is the group ID, the second part is the artifact ID, and the final part is the version information as provided in Maven.

So, it is more like a Maven build script but defined using a Gradle script, which is based on the Groovy language.



Pop Quiz Answers

This appendix contains answers to all the pop quizzes that appear in the chapters. Now, let's have a look at the answers to the respective questions.

Chapter 2, Spring MVC Architecture – Architecting Your Web Store

Questions	Answers
Suppose I have a Spring MVC application for library management called <i>BookPedia</i> and I want to map a web request URL <code>http://localhost:8080/BookPedia/category/fiction</code> to a controller's method—how would you form the <code>@RequestMapping</code> annotation?	2. <code>@RequestMapping("/category/fiction")</code>
What is the request path in the following URL: <code>http://localhost:8080/webstore/?</code>	2. <code>/</code>
Considering the following servlet mapping, identify the possible matching URLs: <pre>@Override protected String[] getServletMappings() { return new String[] { "*.do"}; }</pre>	3. <code>http://localhost:8080/webstore/welcome.do</code>
Considering the following servlet mapping, identify the possible matching URLs: <pre>@Override protected String[] getServletMappings() { return new String[] { "/"}; }</pre>	4. All the above
In order to identify a class as a controller by Spring, what needs to be done?	4. All of the above.

Chapter 3, Control Your Store with Controllers

Questions	Answers
<p>If you imagine a web application called <i>Library</i> with the following request mapping on a Controller class level and in the method level, which is the appropriate request URL to map the request to the <code>books</code> method?</p> <pre>@RequestMapping("/books") public class BookController { ... @RequestMapping(value = "/list") public String books(Model model) { ... } }</pre>	1. <code>http://localhost:8080/library/books/list</code>
<p>Similarly, suppose we have another handler method called <code>bookDetails</code> under <code>BookController</code> as follows, what URL will map to that method?</p> <pre>@RequestMapping public String details(Model model) { ... }</pre>	2. <code>http://localhost:8080/library/books</code>
<p>If we have a web application called <i>webstore</i> with the following request mapping on the Controller class level and in the method level, which is the appropriate request URL?</p> <pre>@RequestMapping("/items") public class ProductController { ... @RequestMapping(value = "/type/{type}", method = RequestMethod.GET) public String productDetails(@PathVariable("type") String productType, Model model) { } }</pre>	2. <code>http://localhost:8080/webstore/items/type/electronics</code>
<p>For the following request mapping annotation, which are the correct methods' signatures to retrieve the path variables?</p> <pre>@RequestMapping(value="/manufacturer/{ manufacturerId}/product/{productId}")</pre>	1. <code>public String productByManufacturer(@PathVariable String manufacturerId, @PathVariable String productId, Model model)</code> 4. <code>public String productByManufacturer(@PathVariable("manufacturerId") String manufacturer, @PathVariable("productId") String product, Model model)</code>
<p>For the following request mapping method signature, which is the appropriate request URL?</p> <pre>@RequestMapping(value = "/products", method = RequestMethod.GET) public String productDetails(@RequestParam String rate, Model model)</pre>	2. <code>http://localhost:8080/webstore/products?rate=400</code>

Chapter 4, Working with Spring Tag Libraries

Questions	Answers
<p>Consider the following data binding customization and identify the possible matching field bindings:</p> <pre>@InitBinder public void initialiseBinder(WebDataBinder binder) { binder.setAllowedFields("unit*"); }</pre>	<p>2. unitPrice 4. united</p>

Chapter 5, Working with View Resolver

Questions	Answers
<p>Consider the following customer Controller:</p> <pre>@Controller("/customers") public class CustomerController { @RequestMapping("/list") public String list(Model model) { return "customers"; } @RequestMapping("/process") public String process(Model model) { // return } }</pre> <p>If I want to redirect the <code>list</code> method from <code>process</code>, how should I form the return statement with the <code>process</code> method?</p>	<p>3. return "redirect:customers/list"</p>
<p>Consider the following resource configuration:</p> <pre>@Override public void addResourceHandlers(ResourceHandlerRegistry registry) { registry.addResourceHandler("/resources/**") .addResourceLocations("/pdf/"); }</pre> <p>Under the <code>pdf</code> directory, if I have a sub-directory such as <code>product/manuals/</code>, which contains a PDF file called <code>manual-P1234.pdf</code>, how can I form the request path to access that PDF file?</p>	<p>2. /resources/product/manuals/manual-P1234.pdf</p>

Chapter 6, Internalize Your Store with Interceptor

Questions	Answers
<p>Consider the following interceptor:</p> <pre>public class SecurityInterceptor extends HandlerInterceptorAdapter{ @Override public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception { // just some code related to after completion } }</pre> <p>Is this <code>SecurityInterceptor</code> class a valid interceptor?</p>	<p>2. It is valid because it extends the <code>HandlerInterceptorAdapter</code> class.</p>
<p>Within the interceptor methods, what is the order of execution?</p>	<p>2. <code>preHandle</code>, <code>postHandle</code>, <code>afterCompletion</code>.</p>

Chapter 7, Incorporating Spring Security

Questions	Answers
<p>Which URL is the Spring Security default authentication handler listening on for the username and password?</p>	<p>1. <code>/login</code></p>
<p>What is the default logout handler URL for Spring Security?</p>	<p>1. <code>/logout</code></p>

Chapter 10, Float Your Application with Web Flow

Questions	Answers
Consider the following web flow registry configuration; it has a single flow definition file, namely validate.xml. How will you form the URL to invoke the flow? <pre>@Bean public FlowDefinitionRegistry flowRegistry() { return getFlowDefinitionRegistryBuilder() .setBasePath("/WEB-INF/flows") .addFlowLocation("/customer/validate.xml", "validateCustomer") .build(); }</pre>	3. http://localhost:8080/webstore/validateCustomer
Consider the following flow invoking URL: http://localhost:8080/webstore/validate?customerId=C1234 In a flow definition file, how will you retrieve the customerId HTTP request parameter?	1. <evaluate expression = "requestParameters.customerId " result = "customerId" />

Chapter 11, Template with Tiles

Questions	Answers
Which of the following statements are true according to Apache Tiles?	4. All of the above.

Module 3: Spring Microservices

Chapter 1: Demystifying Microservices	1
The evolution of microservices	1
What are microservices?	5
Microservices – the honeycomb analogy	8
Principles of microservices	8
Characteristics of microservices	10
Microservices examples	17
Microservices benefits	23
Relationship with other architecture styles	33
Microservice use cases	43
Summary	48
Chapter 2: Building Microservices with Spring Boot	49
Setting up a development environment	49
Developing a RESTful service – the legacy approach	50
Moving from traditional web applications to microservices	55
Using Spring Boot to build RESTful microservices	56
Getting started with Spring Boot	57
Developing the Spring Boot microservice using the CLI	57
Developing the Spring Boot Java microservice using STS	58
Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example	68
What's next?	72
The Spring Boot configuration	73
Changing the default embedded web server	77
Implementing Spring Boot security	77
Enabling cross-origin access for microservices	82
Implementing Spring Boot messaging	83

Developing a comprehensive microservice example	86
Spring Boot actuators	97
Configuring application information	99
Adding a custom health module	99
Documenting microservices	102
Summary	104
Chapter 3: Applying Microservices Concepts	105
Patterns and common design decisions	105
Microservices challenges	139
The microservices capability model	144
Summary	149
Chapter 4: Microservices Evolution – A Case Study	151
Reviewing the microservices capability model	152
Understanding the PSS application	153
Death of the monolith	158
Microservices to the rescue	164
The business case	165
Plan the evolution	165
Migrate modules only if required	187
Target architecture	188
Target implementation view	194
Summary	201
Chapter 5: Scaling Microservices with Spring Cloud	203
Reviewing microservices capabilities	204
Reviewing BrownField's PSS implementation	204
What is Spring Cloud?	205
Setting up the environment for BrownField PSS	210
Spring Cloud Config	211
Feign as a declarative REST client	227
Ribbon for load balancing	229
Eureka for registration and discovery	232
Zuul proxy as the API gateway	244
Streams for reactive microservices	252
Summarizing the BrownField PSS architecture	256
Summary	258
Chapter 6: Autoscaling Microservices	259
Reviewing the microservice capability model	260
Scaling microservices with Spring Cloud	260
Understanding the concept of autoscaling	262
Autoscaling approaches	268

Autoscaling BrownField PSS microservices	272
Summary	282
Chapter 7: Logging and Monitoring Microservices	283
Reviewing the microservice capability model	284
Understanding log management challenges	284
A centralized logging solution	286
The selection of logging solutions	288
Monitoring microservices	297
Data analysis using data lakes	310
Summary	311
Chapter 8: Containerizing Microservices with Docker	313
Reviewing the microservice capability model	314
Understanding the gaps in BrownField PSS microservices	314
What are containers?	316
The difference between VMs and containers	317
The benefits of containers	319
Microservices and containers	320
Introduction to Docker	321
Deploying microservices in Docker	326
Running RabbitMQ on Docker	330
Using the Docker registry	330
Microservices on the cloud	332
Running BrownField services on EC2	332
Updating the life cycle manager	334
The future of containerization – unikernels and hardened security	334
Summary	335
Chapter 9: Managing Dockerized Microservices with Mesos and Marathon	337
Reviewing the microservice capability model	338
The missing pieces	338
Why cluster management is important	340
What does cluster management do?	341
Relationship with microservices	344
Relationship with virtualization	344
Cluster management solutions	344
Cluster management with Mesos and Marathon	348
Implementing Mesos and Marathon for BrownField microservices	353
A place for the life cycle manager	367
The technology metamodel	368
Summary	369

Chapter 10: The Microservices Development Life Cycle	371
Reviewing the microservice capability model	372
The new mantra of lean IT – DevOps	372
Meeting the trio – microservices, DevOps, and cloud	375
Practice points for microservices development	377
Microservices development governance, reference architectures, and libraries	398
Summary	398

Module 3

Spring Microservices

Build scalable microservices with Spring, Docker, and Mesos

1

Demystifying Microservices

Microservices are an architecture style and an approach for software development to satisfy modern business demands. Microservices are not invented; they are more of an evolution from the previous architecture styles.

We will start the chapter by taking a closer look at the evolution of the microservices architecture from the traditional monolithic architectures. We will also examine the definition, concepts, and characteristics of microservices. Finally, we will analyze typical use cases of microservices and establish the similarities and relationships between microservices and other architecture approaches such as **Service Oriented Architecture (SOA)** and Twelve-Factor Apps. Twelve-Factor Apps defines a set of software engineering principles of developing applications targeting the cloud.

In this chapter you, will learn about:

- The evolution of microservices
- The definition of the microservices architecture with examples
- Concepts and characteristics of the microservices architecture
- Typical use cases of the microservices architecture
- The relationship of microservices with SOA and Twelve-Factor Apps

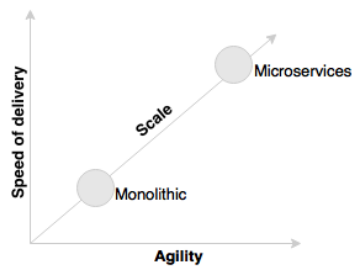
The evolution of microservices

Microservices are one of the increasingly popular architecture patterns next to SOA, complemented by DevOps and cloud. The microservices evolution is greatly influenced by the disruptive digital innovation trends in modern business and the evolution of technologies in the last few years. We will examine these two factors in this section.

Business demand as a catalyst for microservices evolution

In this era of digital transformation, enterprises increasingly adopt technologies as one of the key enablers for radically increasing their revenue and customer base. Enterprises primarily use social media, mobile, cloud, big data, and Internet of Things as vehicles to achieve the disruptive innovations. Using these technologies, enterprises find new ways to quickly penetrate the market, which severely pose challenges to the traditional IT delivery mechanisms.

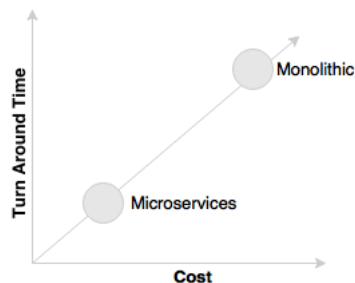
The following graph shows the state of traditional development and microservices against the new enterprise challenges such as agility, speed of delivery, and scale.




[ Microservices promise more agility, speed of delivery, and scale compared to traditional monolithic applications.]

Gone are the days when businesses invested in large application developments with the turnaround time of a few years. Enterprises are no longer interested in developing consolidated applications to manage their end-to-end business functions as they did a few years ago.

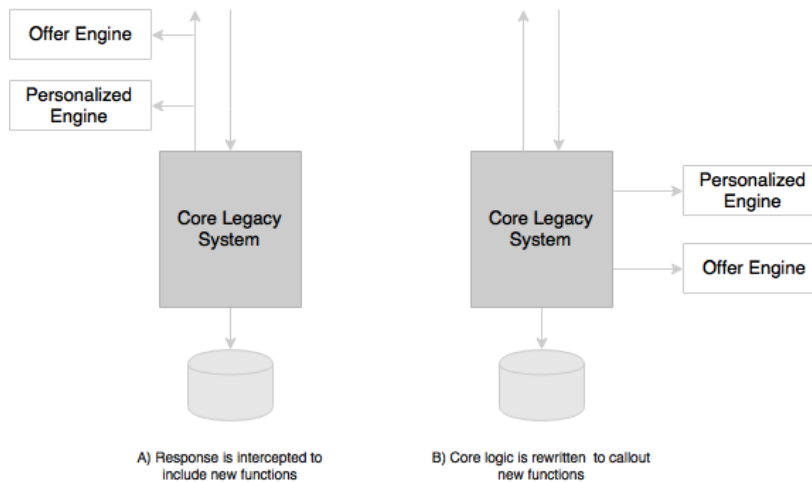
The following graph shows the state of traditional monolithic applications and microservices in comparison with the turnaround time and cost.



 Microservices provide an approach for developing quick and agile applications, resulting in less overall cost.

Today, for instance, airlines or financial institutions do not invest in rebuilding their core mainframe systems as another monolithic monster. Retailers and other industries do not rebuild heavyweight supply chain management applications, such as their traditional ERPs. Focus has shifted to building quick-win point solutions that cater to specific needs of the business in the most agile way possible

Let's take an example of an online retailer running with a legacy monolithic application. If the retailer wants to innovate his/her sales by offering their products personalized to a customer based on the customer's past shopping, preferences, and so on and also wants to enlighten customers by offering products based on their propensity to buy them, they will quickly develop a personalization engine or offers based on their immediate needs and plug them into their legacy application.



As shown in the preceding diagram, rather than investing in rebuilding the core legacy system, this will be either done by passing the responses through the new functions, as shown in the diagram marked **A**, or by modifying the core legacy system to callout these functions as part of the processing, as shown in the diagram marked **B**. These functions are typically written as microservices.

This approach gives organizations a plethora of opportunities to quickly try out new functions with lesser cost in an experimental mode. Businesses can later validate key performance indicators and alter or replace these implementations if required.



Modern architectures are expected to maximize the ability to replace their parts and minimize the cost of replacing their parts. The microservices approach is a means to achieving this.



Technology as a catalyst for the microservices evolution

Emerging technologies have also made us rethink the way we build software systems. For example, a few decades back, we couldn't even imagine a distributed application without a two-phase commit. Later, NoSQL databases made us think differently.

Similarly, these kinds of paradigm shifts in technology have reshaped all the layers of the software architecture.

The emergence of HTML 5 and CSS3 and the advancement of mobile applications repositioned user interfaces. Client-side JavaScript frameworks such as Angular, Ember, React, Backbone, and so on are immensely popular due to their client-side rendering and responsive designs.

With cloud adoptions steamed into the mainstream, **Platform as a Services (PaaS)** providers such as Pivotal CF, AWS, Salesforce.com, IBMs Bluemix, RedHat OpenShift, and so on made us rethink the way we build middleware components. The container revolution created by Docker radically influenced the infrastructure space. These days, an infrastructure is treated as a commodity service.

The integration landscape has also changed with **Integration Platform as a Service (iPaaS)**, which is emerging. Platforms such as Dell Boomi, Informatica, MuleSoft, and so on are examples of iPaaS. These tools helped organizations stretch integration boundaries beyond the traditional enterprise.

NoSQLs have revolutionized the databases space. A few years ago, we had only a few popular databases, all based on relational data modeling principles. We have a long list of databases today: Hadoop, Cassandra, CouchDB, and Neo 4j to name a few. Each of these databases addresses certain specific architectural problems

Imperative architecture evolution

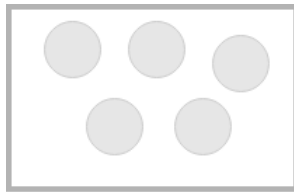
Application architecture has always been evolving alongside demanding business requirements and the evolution of technologies. Architectures have gone through the evolution of age-old mainframe systems to fully abstract cloud services such as AWS Lambda.



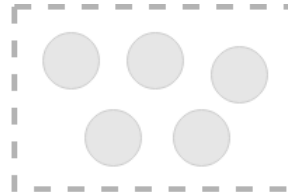
Using AWS Lambda, developers can now drop their "functions" into a fully managed compute service.

Read more about Lambda at: <https://aws.amazon.com/documentation/lambda/>

Different architecture approaches and styles such as mainframes, client server, N-tier, and service-oriented were popular at different timeframes. Irrespective of the choice of architecture styles, we always used to build one or the other forms of monolithic architectures. The microservices architecture evolved as a result of modern business demands such as agility and speed of delivery, emerging technologies, and learning from previous generations of architectures.



**Monolithic
Architecture**



**Microservices
Architecture**

Microservices help us break the boundaries of monolithic applications and build a logically independent smaller system of systems, as shown in the preceding diagram.



If we consider monolithic applications as a set of logical subsystems encompassed with a physical boundary, microservices are a set of independent subsystems with no enclosing physical boundary.


What are microservices?

Microservices are an architecture style used by many organizations today as a game changer to achieve a high degree of agility, speed of delivery, and scale. Microservices give us a way to develop more physically separated modular applications.

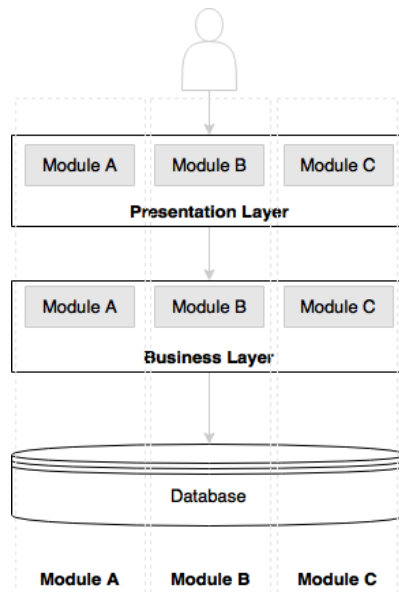
Microservices are not invented. Many organizations such as Netflix, Amazon, and eBay successfully used the divide-and-conquer technique to functionally partition their monolithic applications into smaller atomic units, each performing a single function. These organizations solved a number of prevailing issues they were experiencing with their monolithic applications.

Following the success of these organizations, many other organizations started adopting this as a common pattern to refactor their monolithic applications. Later, evangelists termed this pattern as the microservices architecture.

Microservices originated from the idea of hexagonal architecture coined by Alistair Cockburn. Hexagonal architecture is also known as the Ports and Adapters pattern.

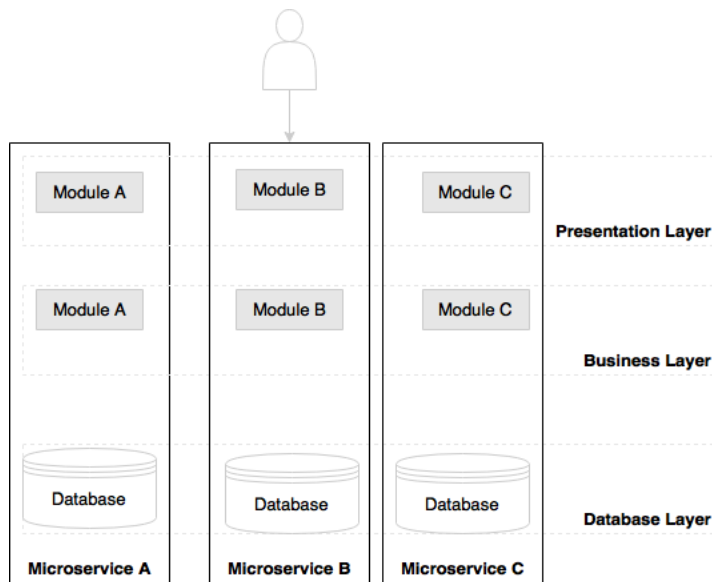
[ Read more about hexagonal architecture at <http://alistair.cockburn.us/Hexagonal+architecture>.]

Microservices are an architectural style or an approach to building IT systems as a set of business capabilities that are autonomous, self-contained, and loosely coupled:



The preceding diagram depicts a traditional N-tier application architecture having a presentation layer, business layer, and database layer. The modules **A**, **B**, and **C** represent three different business capabilities. The layers in the diagram represent a separation of architecture concerns. Each layer holds all three business capabilities pertaining to this layer. The presentation layer has web components of all the three modules, the business layer has business components of all the three modules, and the database hosts tables of all the three modules. In most cases, layers are physically spreadable, whereas modules within a layer are hardwired.

Let's now examine a microservices-based architecture.



As we can note in the preceding diagram, the boundaries are inversed in the microservices architecture. Each vertical slice represents a microservice. Each microservice has its own presentation layer, business layer, and database layer. Microservices are aligned towards business capabilities. By doing so, changes to one microservice do not impact others.

There is no standard for communication or transport mechanisms for microservices. In general, microservices communicate with each other using widely adopted lightweight protocols, such as HTTP and REST, or messaging protocols, such as JMS or AMQP. In specific cases, one might choose more optimized communication protocols, such as Thrift, ZeroMQ, Protocol Buffers, or Avro.

As microservices are more aligned to business capabilities and have independently manageable life cycles, they are the ideal choice for enterprises embarking on DevOps and cloud. DevOps and cloud are two facets of microservices.



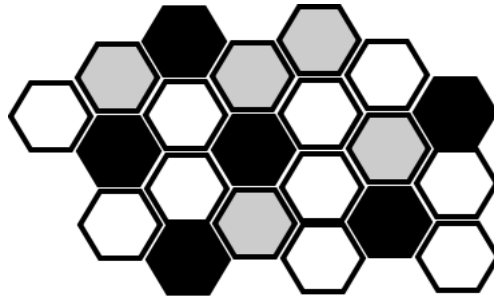
DevOps is an IT realignment to narrow the gap between traditional IT development and operations for better efficiency

Read more about DevOps:

<http://dev2ops.org/2010/02/what-is-devops/>

Microservices – the honeycomb analogy

The honeycomb is an ideal analogy for representing the evolutionary microservices architecture.



In the real world, bees build a honeycomb by aligning hexagonal wax cells. They start small, using different materials to build the cells. Construction is based on what is available at the time of building. Repetitive cells form a pattern and result in a strong fabric structure. Each cell in the honeycomb is independent but also integrated with other cells. By adding new cells, the honeycomb grows organically to a big, solid structure. The content inside each cell is abstracted and not visible outside. Damage to one cell does not damage other cells, and bees can reconstruct these cells without impacting the overall honeycomb.

Principles of microservices

In this section, we will examine some of the principles of the microservices architecture. These principles are a "must have" when designing and developing microservices.

Single responsibility per service

The single responsibility principle is one of the principles defined as part of the SOLID design pattern. It states that a unit should only have one responsibility.

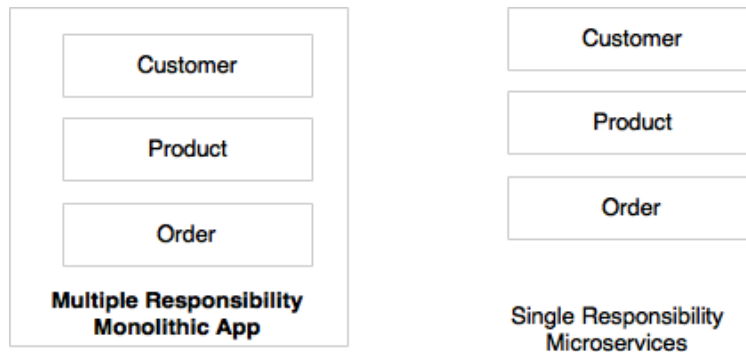


Read more about the SOLID design pattern at:

<http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>



This implies that a unit, either a class, a function, or a service, should have only one responsibility. At no point should two units share one responsibility or one unit have more than one responsibility. A unit with more than one responsibility indicates tight coupling.



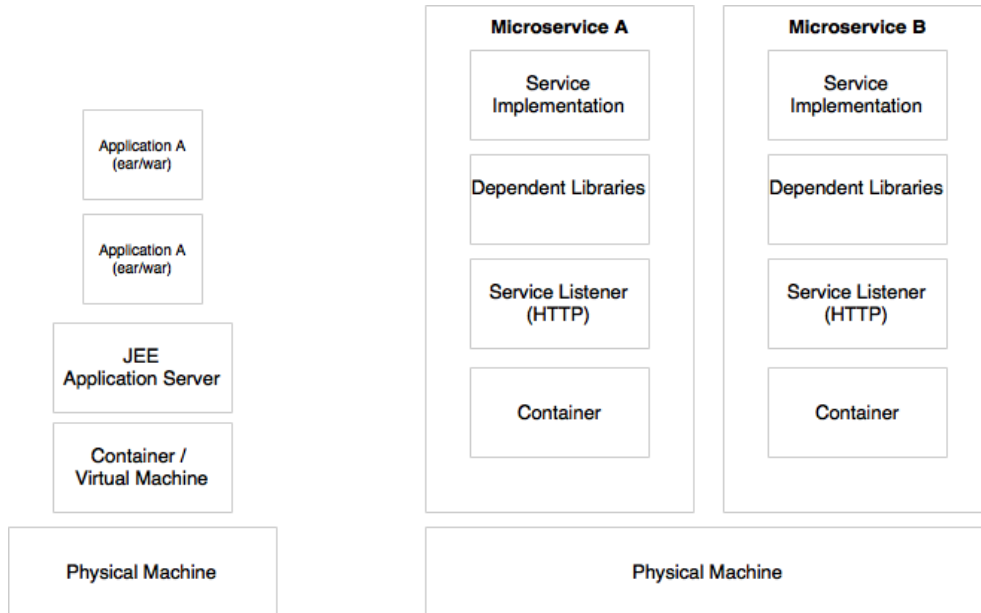
As shown in the preceding diagram, **Customer**, **Product**, and **Order** are different functions of an e-commerce application. Rather than building all of them into one application, it is better to have three different services, each responsible for exactly one business function, so that changes to one responsibility will not impair others. In the preceding scenario, **Customer**, **Product**, and **Order** will be treated as three independent microservices.

Microservices are autonomous

Microservices are self-contained, independently deployable, and autonomous services that take full responsibility of a business capability and its execution. They bundle all dependencies, including library dependencies, and execution environments such as web servers and containers or virtual machines that abstract physical resources.

One of the major differences between microservices and SOA is in their level of autonomy. While most SOA implementations provide service-level abstraction, microservices go further and abstract the realization and execution environment.

In traditional application developments, we build a WAR or an EAR, then deploy it into a JEE application server, such as with JBoss, WebLogic, WebSphere, and so on. We may deploy multiple applications into the same JEE container. In the microservices approach, each microservice will be built as a fat Jar, embedding all dependencies and run as a standalone Java process.



Microservices may also get their own containers for execution, as shown in the preceding diagram. Containers are portable, independently manageable, lightweight runtime environments. Container technologies, such as Docker, are an ideal choice for microservices deployment.

Characteristics of microservices

The microservices definition discussed earlier in this chapter is arbitrary. Evangelists and practitioners have strong but sometimes different opinions on microservices. There is no single, concrete, and universally accepted definition for microservices. However, all successful microservices implementations exhibit a number of common characteristics. Therefore, it is important to understand these characteristics rather than sticking to theoretical definitions. Some of the common characteristics are detailed in this section.

Services are first-class citizens

In the microservices world, services are first-class citizens. Microservices expose service endpoints as APIs and abstract all their realization details. The internal implementation logic, architecture, and technologies (including programming language, database, quality of services mechanisms, and so on) are completely hidden behind the service API.

Moreover, in the microservices architecture, there is no more application development; instead, organizations focus on service development. In most enterprises, this requires a major cultural shift in the way that applications are built.

In a **Customer Profile** microservice, internals such as the data structure, technologies, business logic, and so on are hidden. They aren't exposed or visible to any external entities. Access is restricted through the service endpoints or APIs. For instance, Customer Profile microservices may expose **Register Customer** and **Get Customer** as two APIs for others to interact with.

Characteristics of services in a microservice

As microservices are more or less like a flavor of SOA, many of the service characteristics defined in the SOA are applicable to microservices as well

The following are some of the characteristics of services that are applicable to microservices as well:

- **Service contract:** Similar to SOA, microservices are described through well-defined service contracts. In the microservices world, JSON and REST are universally accepted for service communication. In the case of JSON/REST, there are many techniques used to define service contracts. JSON Schema, WADL, Swagger, and RAML are a few examples.
- **Loose coupling:** Microservices are independent and loosely coupled. In most cases, microservices accept an event as input and respond with another event. Messaging, HTTP, and REST are commonly used for interaction between microservices. Message-based endpoints provide higher levels of decoupling.
- **Service abstraction:** In microservices, service abstraction is not just an abstraction of service realization, but it also provides a complete abstraction of all libraries and environment details, as discussed earlier.
- **Service reuse:** Microservices are course-grained reusable business services. These are accessed by mobile devices and desktop channels, other microservices, or even other systems.

- **Statelessness:** Well-designed microservices are stateless and share nothing with no shared state or conversational state maintained by the services. In case there is a requirement to maintain state, they are maintained in a database, perhaps in memory.
- **Services are discoverable:** Microservices are discoverable. In a typical microservices environment, microservices self-advertise their existence and make themselves available for discovery. When services die, they automatically take themselves out from the microservices ecosystem.
- **Service interoperability:** Services are interoperable as they use standard protocols and message exchange standards. Messaging, HTTP, and so on are used as transport mechanisms. REST/JSON is the most popular method for developing interoperable services in the microservices world. In cases where further optimization is required on communications, other protocols such as Protocol Buffers, Thrift, Avro, or Zero MQ could be used. However, the use of these protocols may limit the overall interoperability of the services.
- **Service composeability:** Microservices are composeable. Service composeability is achieved either through service orchestration or service choreography.



More detail on SOA principles can be found at:

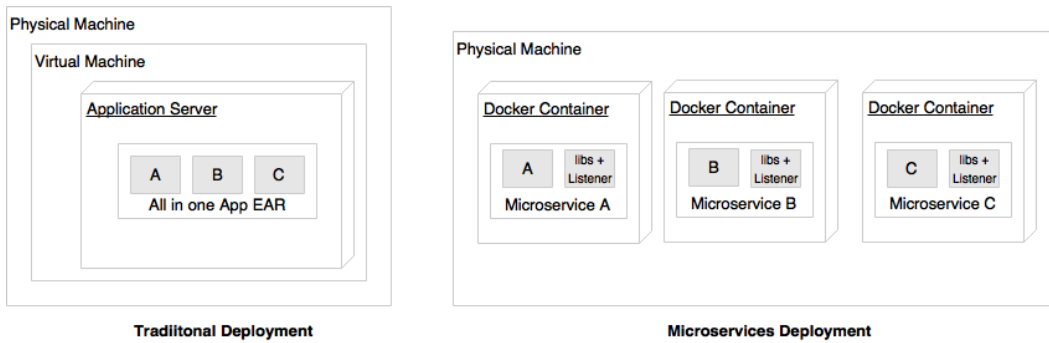
<http://serviceorientation.com/serviceorientation/index>

Microservices are lightweight

Well-designed microservices are aligned to a single business capability, so they perform only one function. As a result, one of the common characteristics we see in most of the implementations are microservices with smaller footprints.

When selecting supporting technologies, such as web containers, we will have to ensure that they are also lightweight so that the overall footprint remains manageable. For example, Jetty or Tomcat are better choices as application containers for microservices compared to more complex traditional application servers such as WebLogic or WebSphere.

Container technologies such as Docker also help us keep the infrastructure footprint as minimal as possible compared to hypervisors such as VMWare or Hyper-V.

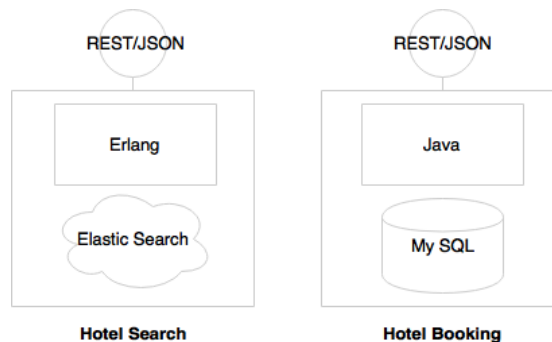


As shown in the preceding diagram, microservices are typically deployed in Docker containers, which encapsulate the business logic and needed libraries. This helps us quickly replicate the entire setup on a new machine or on a completely different hosting environment or even to move across different cloud providers. As there is no physical infrastructure dependency, containerized microservices are easily portable.

Microservices with polyglot architecture

As microservices are autonomous and abstract everything behind service APIs, it is possible to have different architectures for different microservices. A few common characteristics that we see in microservices implementations are:

- Different services use different versions of the same technologies. One microservice may be written on Java 1.7, and another one could be on Java 1.8.
- Different languages are used to develop different microservices, such as one microservice is developed in Java and another one in Scala.
- Different architectures are used, such as one microservice using the Redis cache to serve data, while another microservice could use MySQL as a persistent data store.



In the preceding example, as **Hotel Search** is expected to have high transaction volumes with stringent performance requirements, it is implemented using Erlang. In order to support predictive searching, Elasticsearch is used as the data store. At the same time, **Hotel Booking** needs more ACID transactional characteristics. Therefore, it is implemented using MySQL and Java. The internal implementations are hidden behind service endpoints defined as REST/JSON over HTTP

Automation in a microservices environment

Most of the microservices implementations are automated to a maximum from development to production.

As microservices break monolithic applications into a number of smaller services, large enterprises may see a proliferation of microservices. A large number of microservices is hard to manage until and unless automation is in place. The smaller footprint of microservices also helps us automate the microservices development to the deployment life cycle. In general, microservices are automated end to end – for example, automated builds, automated testing, automated deployment, and elastic scaling.

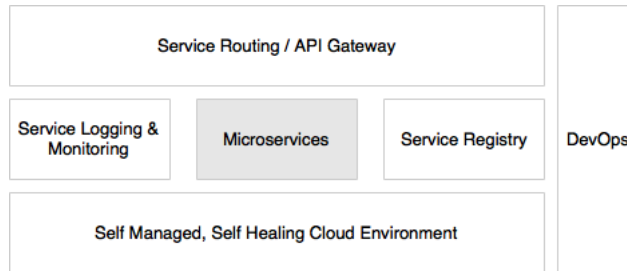


As indicated in the preceding diagram, automations are typically applied during the development, test, release, and deployment phases:

- The development phase is automated using version control tools such as Git together with **Continuous Integration (CI)** tools such as Jenkins, Travis CI, and so on. This may also include code quality checks and automation of unit testing. Automation of a full build on every code check-in is also achievable with microservices.
- The testing phase will be automated using testing tools such as Selenium, Cucumber, and other AB testing strategies. As microservices are aligned to business capabilities, the number of test cases to automate is fewer compared to monolithic applications, hence regression testing on every build also becomes possible.
- Infrastructure provisioning is done through container technologies such as Docker, together with release management tools such as Chef or Puppet, and configuration management tools such as Ansible. Automated deployments are handled using tools such as Spring Cloud, Kubernetes, Mesos, and Marathon.

Microservices with a supporting ecosystem

Most of the large-scale microservices implementations have a supporting ecosystem in place. The ecosystem capabilities include DevOps processes, centralized log management, service registry, API gateways, extensive monitoring, service routing, and flow control mechanisms

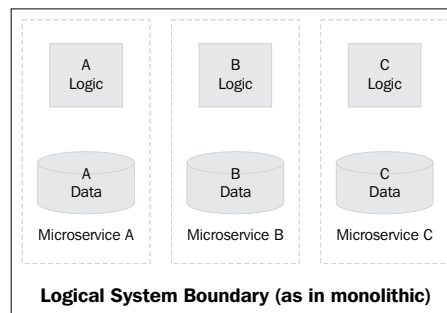


Microservices work well when supporting capabilities are in place, as represented in the preceding diagram.

Microservices are distributed and dynamic

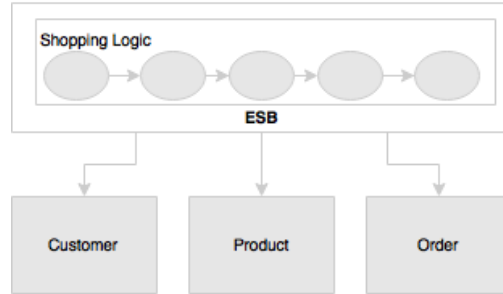
Successful microservices implementations encapsulate logic and data within the service. This results in two unconventional situations: distributed data and logic and decentralized governance.

Compared to traditional applications, which consolidate all logic and data into one application boundary, microservices decentralize data and logic. Each service, aligned to a specific business capability, owns its data and logic



The dotted line in the preceding diagram implies the logical monolithic application boundary. When we migrate this to microservices, each microservice **A**, **B**, and **C** creates its own physical boundaries.

Microservices don't typically use centralized governance mechanisms the way they are used in SOA. One of the common characteristics of microservices implementations is that they do not rely on heavyweight enterprise-level products, such as **Enterprise Service Bus (ESB)**. Instead, the business logic and intelligence are embedded as a part of the services themselves.



A typical SOA implementation is shown in the preceding diagram. Shopping logic is fully implemented in ESB by orchestrating different services exposed by Customer, Order, and Product. In the microservices approach, on the other hand, Shopping itself will run as a separate microservice, which interacts with Customer, Product, and Order in a fairly decoupled way.

SOA implementations heavily rely on static registry and repository configurations to manage services and other artifacts. Microservices bring a more dynamic nature into this. Hence, a static governance approach is seen as an overhead in maintaining up-to-date information. This is why most of the microservices implementations use automated mechanisms to build registry information dynamically from the runtime topologies.

Antifragility, fail fast, and self-healing

Antifragility is a technique successfully experimented at Netflix. It is one of the most powerful approaches to building fail-safe systems in modern software development.



The antifragility concept is introduced by Nassim Nicholas Taleb in his book *Antifragile: Things That Gain from Disorder*.

In the antifragility practice, software systems are consistently challenged. Software systems evolve through these challenges and, over a period of time, get better and better at withstanding these challenges. Amazon's GameDay exercise and Netflix' Simian Army are good examples of such antifragility experiments.

Fail fast is another concept used to build fault-tolerant, resilient systems. This philosophy advocates systems that expect failures versus building systems that never fail. Importance should be given to how quickly the system can fail and if it fails, how quickly it can recover from this failure. With this approach, the focus is shifted from **Mean Time Between Failures (MTBF)** to **Mean Time To Recover (MTTR)**. A key advantage of this approach is that if something goes wrong, it kills itself, and downstream functions aren't stressed.

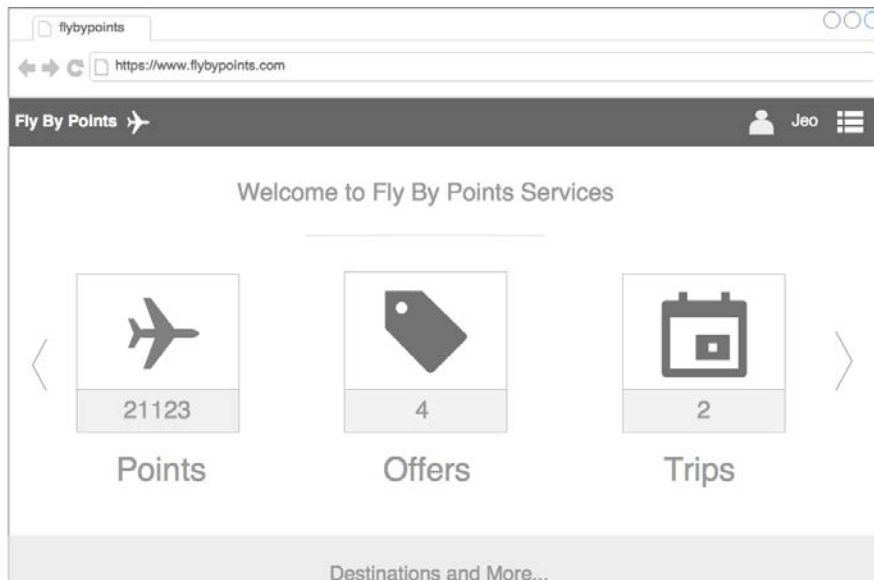
Self-healing is commonly used in microservices deployments, where the system automatically learns from failures and adjusts itself. These systems also prevent future failures.

Microservices examples

There is no "one size fits all" approach when implementing microservices. In this section, different examples are analyzed to crystalize the microservices concept.

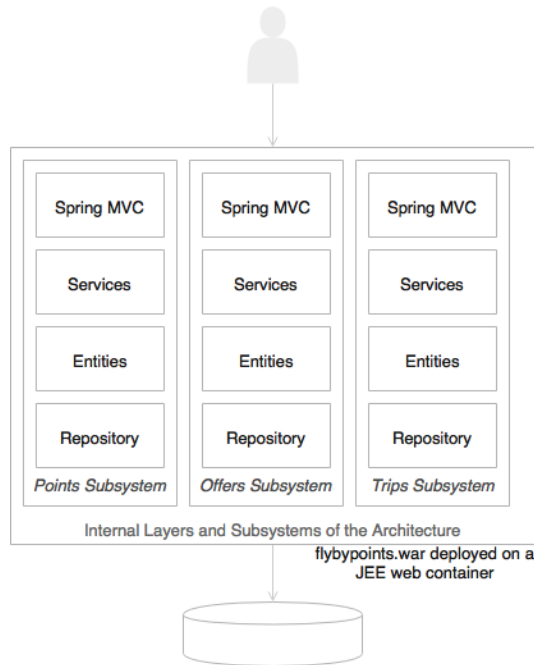
An example of a holiday portal

In the first example, we will review a holiday portal, **Fly By Points**. Fly By Points collects points that are accumulated when a customer books a hotel, flight, or car through the online website. When the customer logs in to the Fly By Points website, he/she is able to see the points accumulated, personalized offers that can be availed of by redeeming the points, and upcoming trips if any.

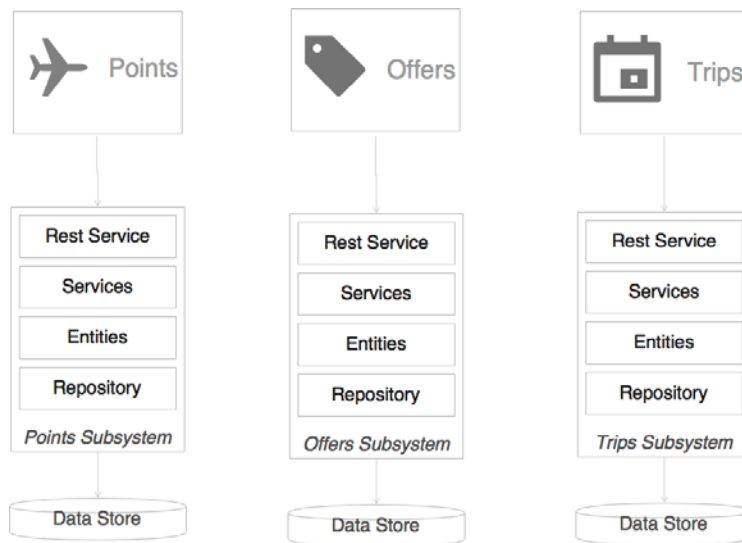


Let's assume that the preceding page is the home page after login. There are two upcoming trips for **Jeo**, four personalized offers, and 21,123 loyalty points. When the user clicks on each of the boxes, the details are queried and displayed.

The holiday portal has a Java Spring-based traditional monolithic application architecture, as shown in the following:



As shown in the preceding diagram, the holiday portal's architecture is web-based and modular, with a clear separation between layers. Following the usual practice, the holiday portal is also deployed as a single WAR file on a web server such as Tomcat. Data is stored on an all-encompassing backing relational database. This is a good fit for the purpose architecture when the complexities are few. As the business grows, the user base expands, and the complexity also increases. This results in a proportional increase in transaction volumes. At this point, enterprises should look to rearchitecting the monolithic application to microservices for better speed of delivery, agility, and manageability.



Examining the simple microservices version of this application, we can immediately note a few things in this architecture:

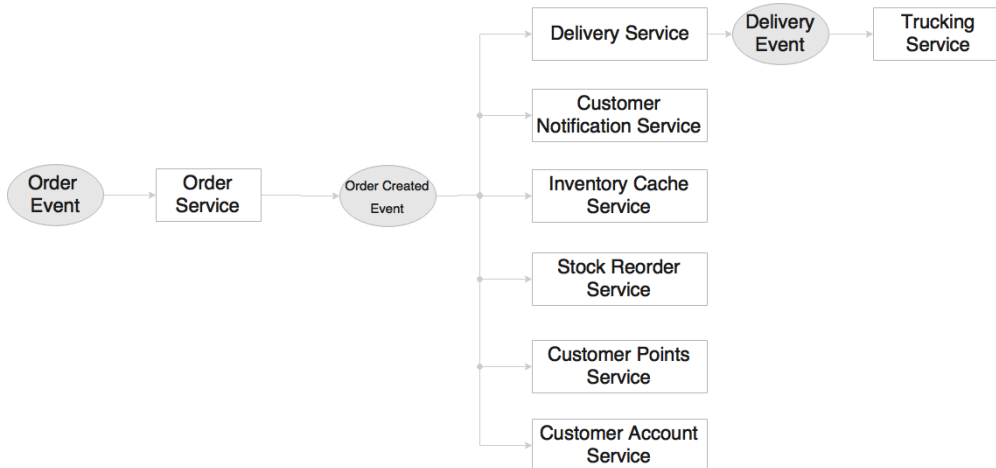
- Each subsystem has now become an independent system by itself, a microservice. There are three microservices representing three business functions: **Trips**, **Offers**, and **Points**. Each one has its internal data store and middle layer. The internal structure of each service remains the same.
- Each service encapsulates its own database as well as its own HTTP listener. As opposed to the previous model, there is no web server or WAR. Instead, each service has its own embedded HTTP listener, such as Jetty, Tomcat, and so on.
- Each microservice exposes a REST service to manipulate the resources/entity that belong to this service.

It is assumed that the presentation layer is developed using a client-side JavaScript MVC framework such as Angular JS. These client-side frameworks are capable of invoking REST calls directly.


When the web page is loaded, all the three boxes, Trips, Offers, and Points will be displayed with details such as points, the number of offers, and the number of trips. This will be done by each box independently making asynchronous calls to the respective backend microservices using REST. There is no dependency between the services at the service layer. When the user clicks on any of the boxes, the screen will be transitioned and will load the details of the item clicked on. This will be done by making another call to the respective microservice.

A microservice-based order management system

Let's examine another microservices example: an online retail website. In this case, we will focus more on the backend services, such as the Order Service which processes the Order Event generated when a customer places an order through the website:



This microservices system is completely designed based on reactive programming practices.

[ Read more on reactive programming at: <http://www.reactivemanifesto.org>]

When an event is published, a number of microservices are ready to kick-start upon receiving the event. Each one of them is independent and does not rely on other microservices. The advantage of this model is that we can keep adding or replacing microservices to achieve specific needs

In the preceding diagram, there are eight microservices shown. The following activities take place upon the arrival of **Order Event**:

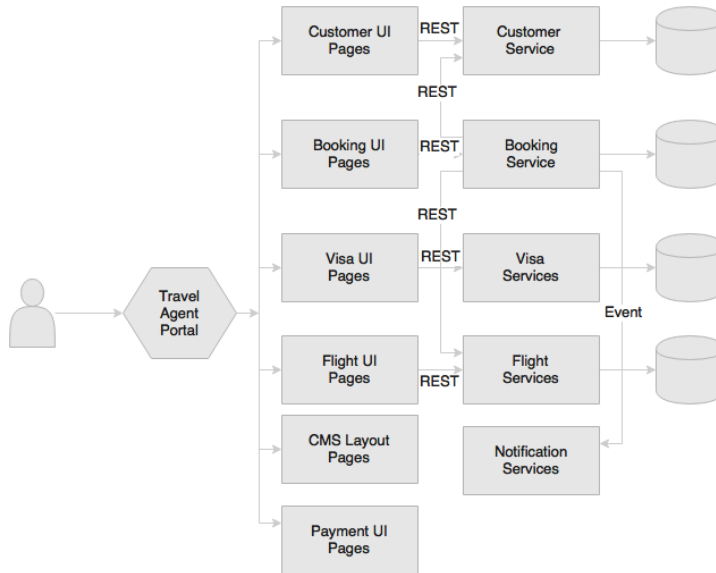
1. Order Service kicks off when Order Event is received. Order Service creates an order and saves the details to its own database.
2. If the order is successfully saved, Order Successful Event is created by Order Service and published.
3. A series of actions take place when Order Successful Event arrives.
4. Delivery Service accepts the event and places Delivery Record to deliver the order to the customer. This, in turn, generates Delivery Event and publishes the event.
5. Trucking Service picks up Delivery Event and processes it. For instance, Trucking Service creates a trucking plan.
6. Customer Notification Service sends a notification to the customer informing the customer that an order is placed.
7. Inventory Cache Service updates the inventory cache with the available product count.
8. Stock Reorder Service checks whether the stock limits are adequate and generates Replenish Event if required.
9. Customer Points Service recalculates the customer's loyalty points based on this purchase.
10. **Customer Account Service** updates the order history in the customer's account.

In this approach, each service is responsible for only one function. Services accept and generate events. Each service is independent and is not aware of its neighborhood. Hence, the neighborhood can organically grow as mentioned in the honeycomb analogy. New services can be added as and when necessary. Adding a new service does not impact any of the existing services.

An example of a travel agent portal

This third example is a simple travel agent portal application. In this example, we will see both synchronous REST calls as well as asynchronous events.

In this case, the portal is just a container application with multiple menu items or links in the portal. When specific pages are requested – for example, when the menu or a link is clicked on – they will be loaded from the specific microservices



When a customer requests a booking, the following events take place internally:

1. The travel agent opens the flight UI, searches for a flight, and identifies the right flight for the customer. Behind the scenes, the flight UI is loaded from the Flight microservice. The flight UI only interacts with its own backend APIs within the Flight microservice. In this case, it makes a REST call to the Flight microservice to load the flights to be displayed
2. The travel agent then queries the customer details by accessing the customer UI. Similar to the flight UI, the customer UI is loaded from the Customer microservice. Actions in the customer UI will invoke REST calls on the Customer microservice. In this case, customer details are loaded by invoking appropriate APIs on the Customer microservice.
3. Then, the travel agent checks the visa details for the customer's eligibility to travel to the selected country. This also follows the same pattern as mentioned in the previous two points.

4. Next, the travel agent makes a booking using the booking UI from the Booking microservice, which again follows the same pattern.
5. The payment pages are loaded from the Payment microservice. In general, the payment service has additional constraints such as PCIDSS compliance (protecting and encrypting data in motion and data at rest). The advantage of the microservices approach is that none of the other microservices need to be considered under the purview of PCIDSS as opposed to the monolithic application, where the complete application comes under the governing rules of PCIDSS. Payment also follows the same pattern as described earlier.
6. Once the booking is submitted, the Booking microservice calls the flight service to validate and update the flight booking. This orchestration is defined as part of the Booking microservice. Intelligence to make a booking is also held within the Booking microservice. As part of the booking process, it also validates, retrieves, and updates the Customer microservice.
7. Finally, the Booking microservice sends the Booking Event, which the Notification service picks up and sends a notification of to the customer.

The interesting factor here is that we can change the user interface, logic, and data of a microservice without impacting any other microservices.

This is a clean and neat approach. A number of portal applications can be built by composing different screens from different microservices, especially for different user communities. The overall behavior and navigation will be controlled by the portal application.

The approach has a number of challenges unless the pages are designed with this approach in mind. Note that the site layouts and static content will be loaded by the **Content Management System (CMS)** as layout templates. Alternately, this could be stored in a web server. The site layout may have fragments of UIs that will be loaded from the microservices at runtime.

Microservices benefits

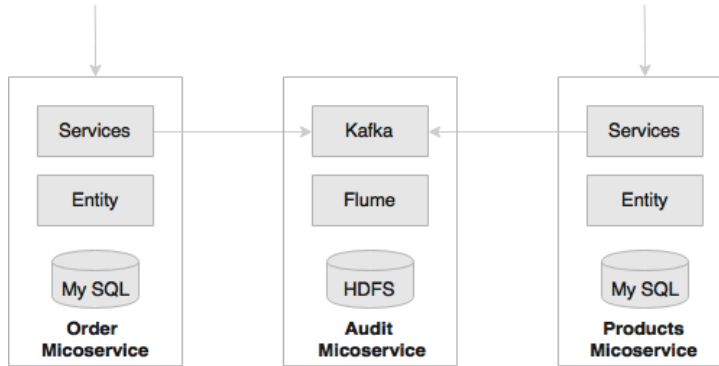
Microservices offer a number of benefits over the traditional multitier, monolithic architectures. This section explains some key benefits of the microservices architecture approach.

Supports polyglot architecture

With microservices, architects and developers can choose fit for purpose architectures and technologies for each microservice. This gives the flexibility to design better-fit solutions in a more cost-effective way

As microservices are autonomous and independent, each service can run with its own architecture or technology or different versions of technologies.

The following shows a simple, practical example of a polyglot architecture with microservices.



There is a requirement to audit all system transactions and record transaction details such as request and response data, the user who initiated the transaction, the service invoked, and so on.

As shown in the preceding diagram, while core services such as the Order and Products microservices use a relational data store, the Audit microservice persists data in Hadoop File System (HDFS). A relational data store is neither ideal nor cost effective in storing large data volumes such as in the case of audit data. In the monolithic approach, the application generally uses a shared, single database that stores Order, Products, and Audit data.

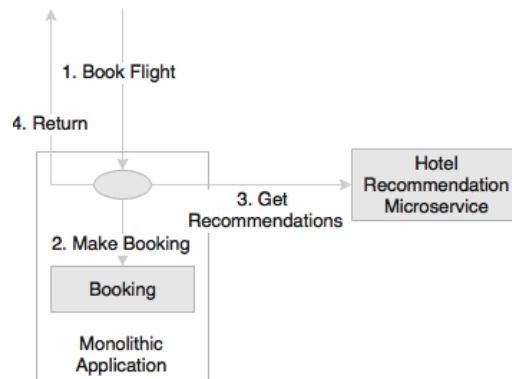
In this example, the audit service is a technical microservice using a different architecture. Similarly, different functional services could also use different architectures.

In another example, there could be a Reservation microservice running on Java 7, while a Search microservice could be running on Java 8. Similarly, an Order microservice could be written on Erlang, whereas a Delivery microservice could be on the Go language. None of these are possible with a monolithic architecture.

Enabling experimentation and innovation

Modern enterprises are thriving towards quick wins. Microservices are one of the key enablers for enterprises to do disruptive innovation by offering the ability to experiment and fail fast.

As services are fairly simple and smaller in size, enterprises can afford to experiment new processes, algorithms, business logics, and so on. With large monolithic applications, experimentation was not easy; nor was it straightforward or cost effective. Businesses had to spend huge money to build or change an application to try out something new. With microservices, it is possible to write a small microservice to achieve the targeted functionality and plug it into the system in a reactive style. One can then experiment with the new function for a few months, and if the new microservice does not work as expected, we can change or replace it with another one. The cost of change will be considerably less compared to that of the monolithic approach.



In another example of an airline booking website, the airline wants to show personalized hotel recommendations in their booking page. The recommendations must be displayed on the booking confirmation page

As shown in the preceding diagram, it is convenient to write a microservice that can be plugged into the monolithic applications booking flow rather than incorporating this requirement in the monolithic application itself. The airline may choose to start with a simple recommendation service and keep replacing it with newer versions till it meets the required accuracy.

Elastically and selectively scalable

As microservices are smaller units of work, they enable us to implement selective scalability.

Scalability requirements may be different for different functions in an application. A monolithic application, packaged as a single WAR or an EAR, can only be scaled as a whole. An I/O-intensive function when streamed with high velocity data could easily bring down the service levels of the entire application.

In the case of microservices, each service could be independently scaled up or down. As scalability can be selectively applied at each service, the cost of scaling is comparatively less with the microservices approach.

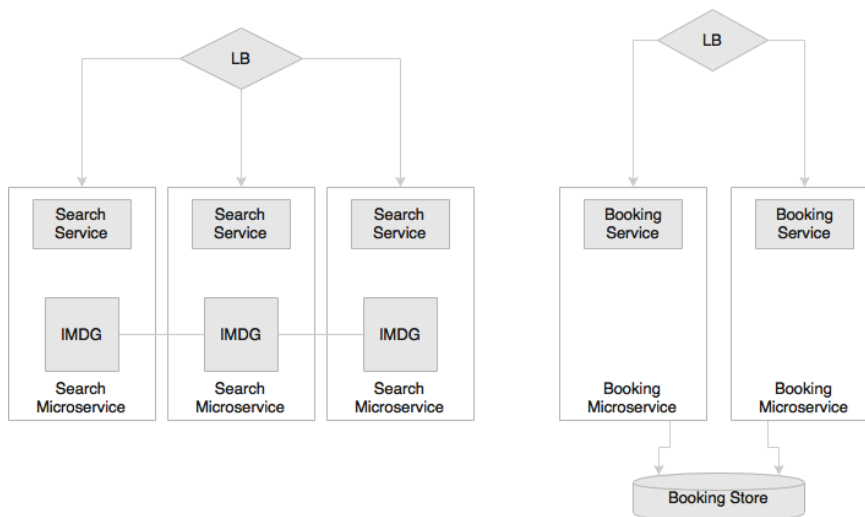
In practice, there are many different ways available to scale an application and is largely subject to the architecture and behavior of the application. **Scale Cube** defines primarily three approaches to scaling an application

- Scaling the x axis by horizontally cloning the application
- Scaling the y axis by splitting different functionality
- Scaling the z axis by partitioning or sharding the data

[ Read more about Scale Cube in the following site: <http://theartofscalability.com/>]

When y axis scaling is applied to monolithic applications, it breaks the monolithic to smaller units aligned with business functions. Many organizations successfully applied this technique to move away from monolithic applications. In principle, the resulting units of functions are in line with the microservices characteristics.

For instance, in a typical airline website, statistics indicate that the ratio of flight searching to flight booking could be as high as 500:1. This means one booking transaction for every 500 search transactions. In this scenario, the search needs 500 times more scalability than the booking function. This is an ideal use case for selective scaling.



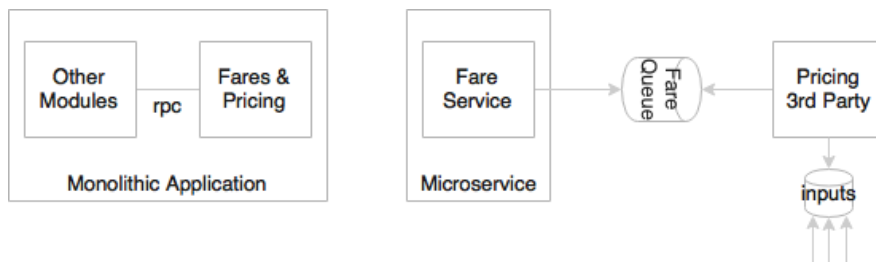
The solution is to treat search requests and booking requests differently. With a monolithic architecture, this is only possible with *z* scaling in the scale cube. However, this approach is expensive because in the *z* scale, the entire code base is replicated.

In the preceding diagram, Search and Booking are designed as different microservices so that Search can be scaled differently from Booking. In the diagram, Search has three instances, and Booking has two instances. Selective scalability is not limited to the number of instances, as shown in the diagram, but also in the way in which the microservices are architected. In the case of Search, an **in-memory data grid (IMDG)** such as Hazelcast can be used as the data store. This will further increase the performance and scalability of Search. When a new Search microservice instance is instantiated, an additional IMDG node is added to the IMDG cluster. Booking does not require the same level of scalability. In the case of Booking, both instances of the Booking microservice are connected to the same instance of the database.

Allowing substitution

Microservices are self-contained, independent deployment modules enabling the substitution of one microservice with another similar microservice.

Many large enterprises follow buy-versus-build policies to implement software systems. A common scenario is to build most of the functions in house and buy certain niche capabilities from specialists outside. This poses challenges in traditional monolithic applications as these application components are highly cohesive. Attempting to plug in third-party solutions to the monolithic applications results in complex integrations. With microservices, this is not an afterthought. Architecturally, a microservice can be easily replaced by another microservice developed either in-house or even extended by a microservice from a third party.



A pricing engine in the airline business is complex. Fares for different routes are calculated using complex mathematical formulas known as the pricing logic. Airlines may choose to buy a pricing engine from the market instead of building the product in house. In the monolithic architecture, Pricing is a function of Fares and Booking. In most cases Pricing, Fares, and Booking are hardwired, making it almost impossible to detach.

In a well-designed microservices system, Booking, Fares, and Pricing would be independent microservices. Replacing the Pricing microservice will have only a minimal impact on any other services as they are all loosely coupled and independent. Today, it could be a third-party service; tomorrow, it could be easily substituted by another third-party or home-grown service.

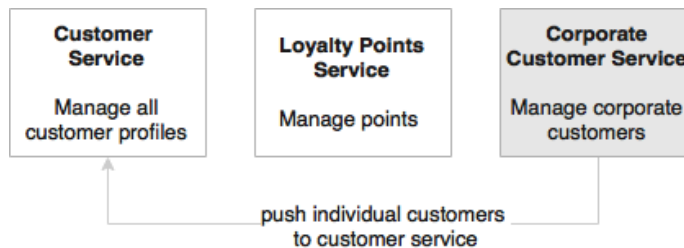
Enabling to build organic systems

Microservices help us build systems that are organic in nature. This is significantly important when migrating monolithic systems gradually to microservices.

Organic systems are systems that grow laterally over a period of time by adding more and more functions to it. In practice, an application grows unimaginably large in its lifespan, and in most cases, the manageability of the application reduces dramatically over this same period of time.

Microservices are all about independently manageable services. This enable us to keep adding more and more services as the need arises with minimal impact on the existing services. Building such systems does not need huge capital investments. Hence, businesses can keep building as part of their operational expenditure.

A loyalty system in an airline was built years ago, targeting individual passengers. Everything was fine until the airline started offering loyalty benefits to their corporate customers. Corporate customers are individuals grouped under corporations. As the current systems core data model is flat, targeting individuals, the corporate environment needs a fundamental change in the core data model, and hence huge reworking, to incorporate this requirement.



As shown in the preceding diagram, in a microservices-based architecture, customer information would be managed by the Customer microservice and loyalty by the Loyalty Points microservice.

In this situation, it is easy to add a new Corporate Customer microservice to manage corporate customers. When a corporation is registered, individual members will be pushed to the Customer microservice to manage them as usual. The Corporate Customer microservice provides a corporate view by aggregating data from the Customer microservice. It will also provide services to support corporate-specific business rules. With this approach, adding new services will have only a minimal impact on the existing services.

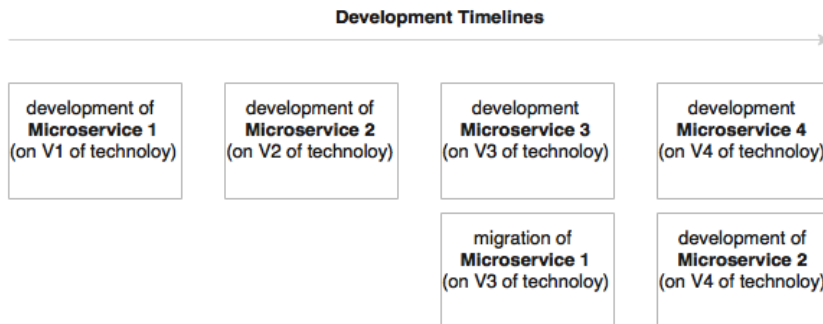
Helping reducing technology debt

As microservices are smaller in size and have minimal dependencies, they allow the migration of services that use end-of-life technologies with minimal cost.

Technology changes are one of the barriers in software development. In many traditional monolithic applications, due to the fast changes in technologies, today's next-generation applications could easily become legacy even before their release to production. Architects and developers tend to add a lot of protection against technology changes by adding layers of abstractions. However, in reality, this approach does not solve the issue but, instead, results in over-engineered systems. As technology upgrades are often risky and expensive with no direct returns to business, the business may not be happy to invest in reducing the technology debt of the applications.

With microservices, it is possible to change or upgrade technology for each service individually rather than upgrading an entire application.

Upgrading an application with, for instance, five million lines written on EJB 1.1 and Hibernate to the Spring, JPA, and REST services is almost similar to rewriting the entire application. In the microservices world, this could be done incrementally.



As shown in the preceding diagram, while older versions of the services are running on old versions of technologies, new service developments can leverage the latest technologies. The cost of migrating microservices with end-of-life technologies is considerably less compared to enhancing monolithic applications.

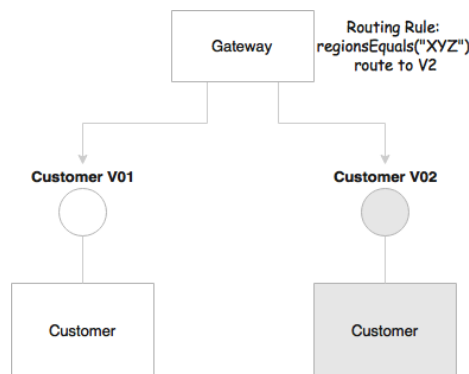
Allowing the coexistence of different versions

As microservices package the service runtime environment along with the service itself, this enables having multiple versions of the service to coexist in the same environment.

There will be situations where we will have to run multiple versions of the same service at the same time. Zero downtime promote, where one has to gracefully switch over from one version to another, is one example of a such a scenario as there will be a time window where both services will have to be up and running simultaneously. With monolithic applications, this is a complex procedure because upgrading new services in one node of the cluster is cumbersome as, for instance, this could lead to class loading issues. A canary release, where a new version is only released to a few users to validate the new service, is another example where multiple versions of the services have to coexist.

With microservices, both these scenarios are easily manageable. As each microservice uses independent environments, including service listeners such as Tomcat or Jetty embedded, multiple versions can be released and gracefully transitioned without many issues. When consumers look up services, they look for specific versions of services. For example, in a canary release, a new user interface is released to user A. When user A sends a request to the microservice, it looks up the canary release version, whereas all other users will continue to look up the last production version.

Care needs to be taken at the database level to ensure the database design is always backward compatible to avoid breaking the changes.



As shown in the preceding diagram, version 1 and 2 of the **Customer** service can coexist as they are not interfering with each other, given their respective deployment environments. Routing rules can be set at the gateway to divert traffic to specific instances, as shown in the diagram. Alternatively, clients can request specific versions as part of the request itself. In the diagram, the gateway selects the version based on the region from which the request is originated.

Supporting the building of self-organizing systems

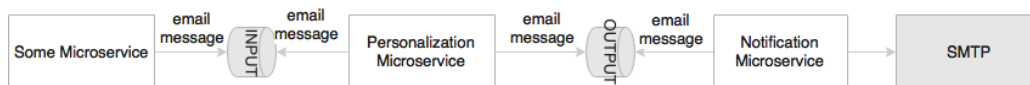
Microservices help us build self-organizing systems. A self-organizing system support will automate deployment, be resilient, and exhibit self-healing and self-learning capabilities.

In a well-architected microservices system, a service is unaware of other services. It accepts a message from a selected queue and processes it. At the end of the process, it may send out another message, which triggers other services. This allows us to drop any service into the ecosystem without analyzing the impact on the overall system. Based on the input and output, the service will self-organize into the ecosystem. No additional code changes or service orchestration is required. There is no central brain to control and coordinate the processes.

Imagine an existing notification service that listens to an **INPUT** queue and sends notifications to an **SMTP** server, as shown in the following figure



Let's assume, later, a personalization engine, responsible for changing the language of the message to the customer's native language, needs to be introduced to personalize messages before sending them to the customer, the personalization engine is responsible for changing the language of the message to the customer's native language.



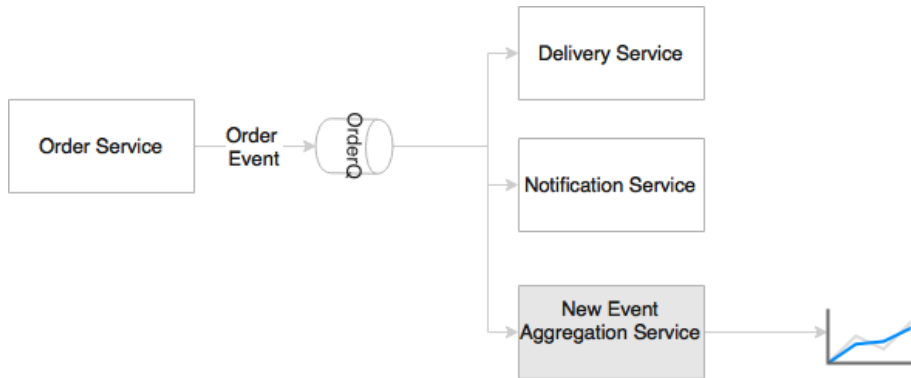
With microservices, a new personalization microservice will be created to do this job. The input queue will be configured as INPUT in an external configuration server, and the personalization service will pick up the messages from the INPUT queue (earlier, this was used by the notification service) and send the messages to the OUTPUT queue after completing process. The notification services input queue will then send to OUTPUT. From the very next moment onward, the system automatically adopts this new message flow

Supporting event-driven architecture

Microservices enable us to develop transparent software systems. Traditional systems communicate with each other through native protocols and hence behave like a black box application. Business events and system events, unless published explicitly, are hard to understand and analyze. Modern applications require data for business analysis, to understand dynamic system behaviors, and analyze market trends, and they also need to respond to real-time events. Events are useful mechanisms for data extraction.

A well-architected microservice always works with events for both input and output. These events can be tapped by any service. Once extracted, events can be used for a variety of use cases.

For example, the business wants to see the velocity of orders categorized by product type in real time. In a monolithic system, we need to think about how to extract these events. This may impose changes in the system.



In the microservices world, **Order Event** is already published whenever an order is created. This means that it is just a matter of adding a new service to subscribe to the same topic, extract the event, perform the requested aggregations, and push another event for the dashboard to consume.

Enabling DevOps

Microservices are one of the key enablers of DevOps. DevOps is widely adopted as a practice in many enterprises, primarily to increase the speed of delivery and agility. A successful adoption of DevOps requires cultural changes, process changes, as well as architectural changes. DevOps advocates to have agile development, high-velocity release cycles, automatic testing, automatic infrastructure provisioning, and automated deployment.

Automating all these processes is extremely hard to achieve with traditional monolithic applications. Microservices are not the ultimate answer, but microservices are at the center stage in many DevOps implementations. Many DevOps tools and techniques are also evolving around the use of microservices.

Consider a monolithic application that takes hours to complete a full build and 20 to 30 minutes to start the application; one can see that this kind of application is not ideal for DevOps automation. It is hard to automate continuous integration on every commit. As large, monolithic applications are not automation friendly, continuous testing and deployments are also hard to achieve.

On the other hand, small footprint microservices are more automation-friendly and therefore can more easily support these requirements.

Microservices also enable smaller, focused agile teams for development. Teams will be organized based on the boundaries of microservices.

Relationship with other architecture styles

Now that we have seen the characteristics and benefits of microservices, in this section, we will explore the relationship of microservices with other closely related architecture styles such as SOA and Twelve-Factor Apps.

Relations with SOA

SOA and microservices follow similar concepts. Earlier in this chapter, we discussed that microservices are evolved from SOA, and many service characteristics are common in both approaches.

However, are they the same or are they different?

As microservices are evolved from SOA, many characteristics of microservices are similar to SOA. Let's first examine the definition of SO

The definition of SOA from *The Open Group* consortium is as follows:

"Service-Oriented Architecture (SOA) is an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services."

A service:

Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)

It is self-contained.

It may be composed of other services.

It is a "black box" to consumers of the service."

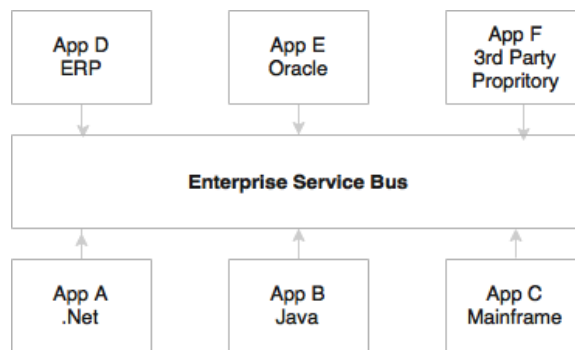
We observed similar aspects in microservices as well. So, in what way are microservices different? The answer is: it depends.

The answer to the previous question could be yes or no, depending upon the organization and its adoption of SOA. SOA is a broader term, and different organizations approached SOA differently to solve different organizational problems. The difference between microservices and SOA is in a way based on how an organization approaches SOA.

In order to get clarity, a few cases will be examined.

Service-oriented integration

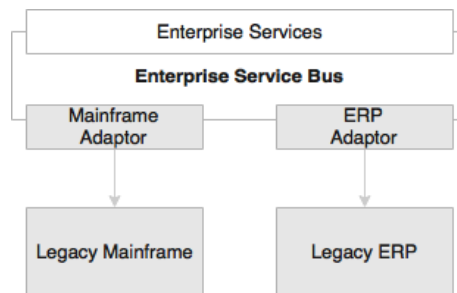
Service-oriented integration refers to a service-based integration approach used by many organizations.



Many organizations would have used SOA primarily to solve their integration complexities, also known as integration spaghetti. Generally, this is termed as **Service-Oriented Integration (SOI)**. In such cases, applications communicate with each other through a common integration layer using standard protocols and message formats such as SOAP/XML-based web services over HTTP or JMS. These types of organizations focus on **Enterprise Integration Patterns (EIP)** to model their integration requirements. This approach strongly relies on heavyweight ESB such as TIBCO Business Works, WebSphere ESB, Oracle ESB, and the likes. Most ESB vendors also packed a set of related products such as rules engines, business process management engines, and so on as an SOA suite. Such organizations' integrations are deeply rooted into their products. They either write heavy orchestration logic in the ESB layer or the business logic itself in the service bus. In both cases, all enterprise services are deployed and accessed via ESB. These services are managed through an enterprise governance model. For such organizations, microservices are altogether different from SOA.

Legacy modernization

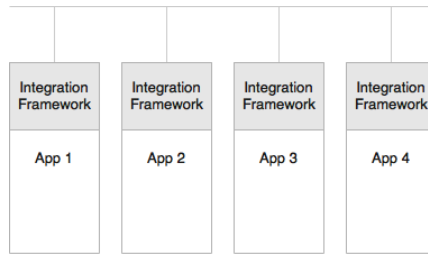
SOA is also used to build service layers on top of legacy applications.



Another category of organizations would use SOA in transformation projects or legacy modernization projects. In such cases, the services are built and deployed in the ESB layer connecting to backend systems using ESB adapters. For these organizations, microservices are different from SOA.

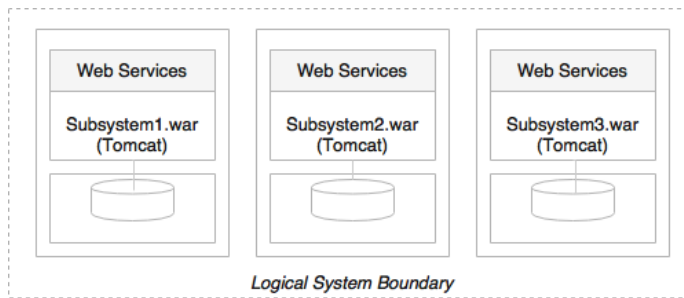
Service-oriented application

Some organizations adopt SOA at an application level.



In this approach, lightweight integration frameworks, such as Apache Camel or Spring Integration, are embedded within applications to handle service-related cross-cutting capabilities such as protocol mediation, parallel execution, orchestration, and service integration. As some of the lightweight integration frameworks have native Java object support, such applications would even use native **Plain Old Java Objects (POJO)** services for integration and data exchange between services. As a result, all services have to be packaged as one monolithic web archive. Such organizations could see microservices as the next logical step of their SOA.

Monolithic migration using SOA



The last possibility is transforming a monolithic application into smaller units after hitting the breaking point with the monolithic system. They would break the application into smaller, physically deployable subsystems, similar to the *y* axis scaling approach explained earlier, and deploy them as web archives on web servers or as JARs deployed on some home-grown containers. These subsystems as service would use web services or other lightweight protocols to exchange data between services. They would also use SOA and service design principles to achieve this. For such organizations, they may tend to think that microservices are the same old wine in a new bottle.

Relations with Twelve-Factor apps

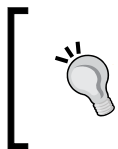
Cloud computing is one of the rapidly evolving technologies. Cloud computing promises many benefits, such as a cost advantage, speed, agility, flexibility, and elasticity. There are many cloud providers offering different services. They lower the cost models to make it more attractive to the enterprises. Different cloud providers such as AWS, Microsoft, Rackspace, IBM, Google, and so on use different tools, technologies, and services. On the other hand, enterprises are aware of this evolving battlefield and, therefore, they are looking for options for de-risking from lockdown to a single vendor.

Many organizations do lift and shift their applications to the cloud. In such cases, the applications may not realize all the benefits promised by cloud platforms. Some applications need to undergo overhaul, whereas some may need minor tweaking before moving to cloud. This by and large depends upon how the application is architected and developed.

For example, if the application has its production database server URLs hardcoded as part of the applications WAR, it needs to be modified before moving the application to cloud. In the cloud, the infrastructure is transparent to the application, and especially, the physical IP addresses cannot be assumed.

How do we ensure that an application, or even microservices, can run seamlessly across multiple cloud providers and take advantages of cloud services such as elasticity?

It is important to follow certain principles while developing cloud native applications.

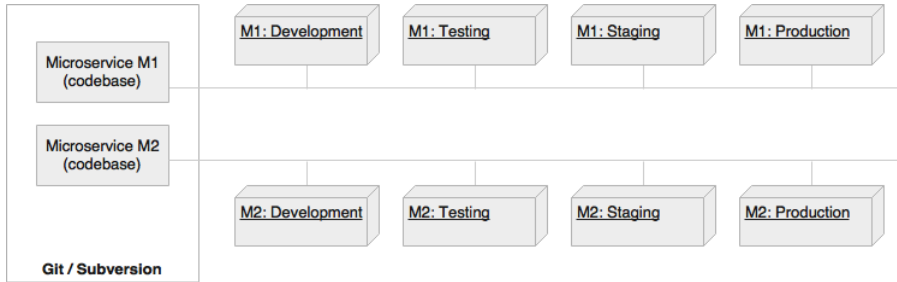


Cloud native is a term used for developing applications that can work efficiently in a cloud environment, understanding and utilizing cloud behaviors such as elasticity, utilization based charging, fail aware, and so on.

Twelve-Factor App, forwarded by Heroku, is a methodology describing the characteristics expected from modern cloud-ready applications. Twelve-Factor App is equally applicable for microservices as well. Hence, it is important to understand Twelve-Factor App.

A single code base

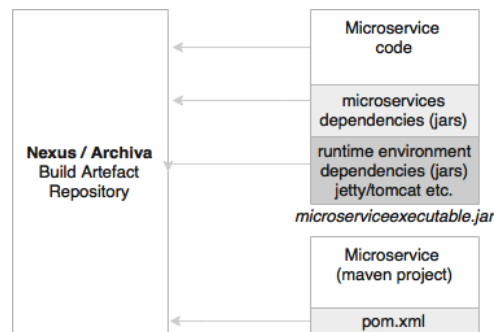
The code base principle advises that each application has a single code base. There can be multiple instances of deployment of the same code base, such as development, testing, and production. Code is typically managed in a source control system such as Git, Subversion, and so on.



Extending the same philosophy for microservices, each microservice should have its own code base, and this code base is not shared with any other microservice. It also means that one microservice has exactly one code base.

Bundling dependencies

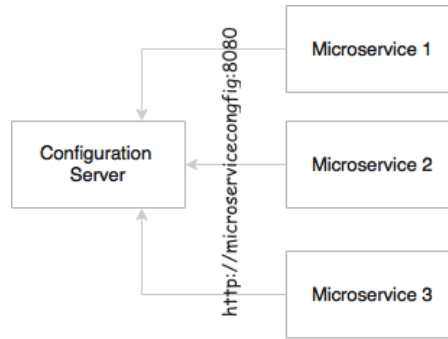
As per this principle, all applications should bundle their dependencies along with the application bundle. With build tools such as Maven and Gradle, we explicitly manage dependencies in a `pom.xml` or the `.gradle` file and link them using a central build artifact repository such as Nexus or Archiva. This ensures that the versions are managed correctly. The final executables will be packaged as a WAR file or a executable JAR file, embedding all the dependencies



In the context of microservices, this is one of the fundamental principles to be followed. Each microservice should bundle all the required dependencies and execution libraries such as the HTTP listener and so on in the final executable bundle.

Externalizing configurations

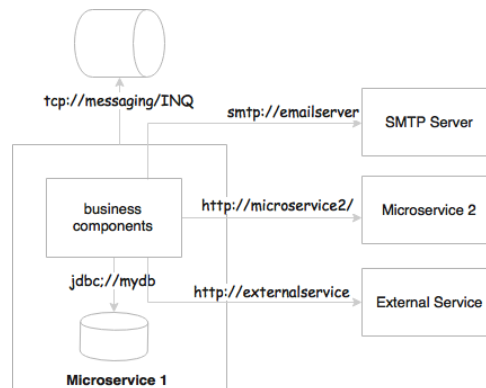
This principle advises the externalization of all configuration parameters from the code. An application's configuration parameters vary between environments, such as support to the e-mail IDs or URL of an external system, username, passwords, queue name, and so on. These will be different for development, testing, and production. All service configurations should be externalized



The same principle is obvious for microservices as well. The microservices configuration parameters should be loaded from an external source. This will also help to automate the release and deployment process as the only difference between these environments is the configuration parameters

Backing services are addressable

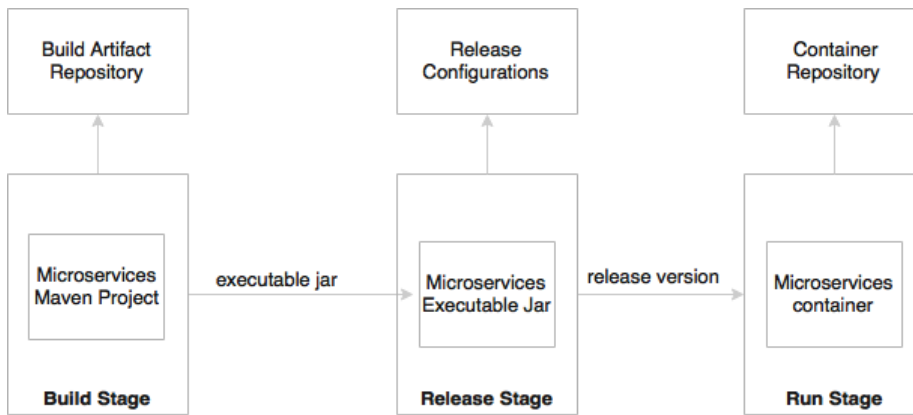
All backing services should be accessible through an addressable URL. All services need to talk to some external resources during the life cycle of their execution. For example, they could be listening or sending messages to a messaging system, sending an e-mail, persisting data to database, and so on. All these services should be reachable through a URL without complex communication requirements.



In the microservices world, microservices either talk to a messaging system to send or receive messages, or they could accept or send messages to other service APIs. In a regular case, these are either HTTP endpoints using REST and JSON or TCP- or HTTP-based messaging endpoints.

Isolation between build, release, and run

This principle advocates a strong isolation between the build, release, and run stages. The build stage refers to compiling and producing binaries by including all the assets required. The release stage refers to combining binaries with environment-specific configuration parameters. The run stage refers to running application on specific execution environment. The pipeline is unidirectional, so it is not possible to propagate changes from the run stages back to the build stage. Essentially, it also means that it is not recommended to do specific builds for production; rather, it has to go through the pipeline.



In microservices, the build will create executable JAR files, including the service runtime such as an HTTP listener. During the release phase, these executables will be combined with release configurations such as production URLs and so on and create a release version, most probably as a container similar to Docker. In the run stage, these containers will be deployed on production via a container scheduler.

Stateless, shared nothing processes

This principle suggests that processes should be stateless and share nothing. If the application is stateless, then it is fault tolerant and can be scaled out easily.

All microservices should be designed as stateless functions. If there is any requirement to store a state, it should be done with a backing database or in an in-memory cache.

Exposing services through port bindings

A Twelve-Factor application is expected to be self-contained. Traditionally, applications are deployed to a server: a web server or an application server such as Apache Tomcat or JBoss. A Twelve-Factor application does not rely on an external web server. HTTP listeners such as Tomcat or Jetty have to be embedded in the service itself.

Port binding is one of the fundamental requirements for microservices to be autonomous and self-contained. Microservices embed service listeners as a part of the service itself.

Concurrency to scale out

This principle states that processes should be designed to scale out by replicating the processes. This is in addition to the use of threads within the process.

In the microservices world, services are designed to scale out rather than scale up. The x axis scaling technique is primarily used for a scaling service by spinning up another identical service instance. The services can be elastically scaled or shrunk based on the traffic flow. Further to this, microservices may make use of parallel processing and concurrency frameworks to further speed up or scale up the transaction processing.

Disposability with minimal overhead

This principle advocates building applications with minimal startup and shutdown times with graceful shutdown support. In an automated deployment environment, we should be able to bring up or bring down instances as quick as possible. If the application's startup or shutdown takes considerable time, it will have an adverse effect on automation. The startup time is proportionally related to the size of the application. In a cloud environment targeting auto-scaling, we should be able to spin up new instance quickly. This is also applicable when promoting new versions of services.

In the microservices context, in order to achieve full automation, it is extremely important to keep the size of the application as thin as possible, with minimal startup and shutdown time. Microservices also should consider a lazy loading of objects and data.

Development and production parity

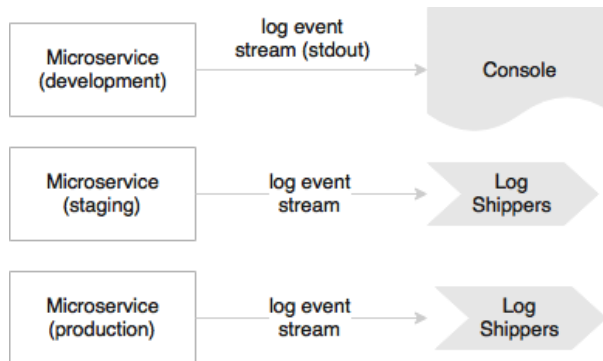
This principle states the importance of keeping development and production environments as identical as possible. For example, let's consider an application with multiple services or processes, such as a job scheduler service, cache services, and one or more application services. In a development environment, we tend to run all of them on a single machine, whereas in production, we will facilitate independent machines to run each of these processes. This is primarily to manage the cost of infrastructure. The downside is that if production fails, there is no identical environment to re-produce and fix the issues

Not only is this principle valid for microservices, but it is also applicable to any application development.

Externalizing logs

A Twelve-Factor application never attempts to store or ship log files. In a cloud, it is better to avoid local I/Os. If the I/Os are not fast enough in a given infrastructure, it could create a bottleneck. The solution to this is to use a centralized logging framework. Splunk, Greylog, Logstash, Logplex, and Loggly are some examples of log shipping and analysis tools. The recommended approach is to ship logs to a central repository by tapping the logback appenders and write to one of the shippers' endpoints.

In a microservices ecosystem, this is very important as we are breaking a system into a number of smaller services, which could result in decentralized logging. If they store logs in a local storage, it would be extremely difficult to correlate logs between services.



In development, the microservice may direct the log stream to `stdout`, whereas in production, these streams will be captured by the log shippers and sent to a central log service for storage and analysis.

Package admin processes

Apart from application services, most applications provide admin tasks as well. This principle advises to use the same release bundle as well as an identical environment for both application services and admin tasks. Admin code should also be packaged along with the application code.

Not only is this principle valid for microservices, but also it is applicable to any application development.

Microservice use cases

A microservice is not a silver bullet and will not solve all the architectural challenges of today's world. There is no hard-and-fast rule or rigid guideline on when to use microservices.

Microservices may not fit in each and every use case. The success of microservices largely depends on the selection of use cases. The first and the foremost activity is to do a litmus test of the use case against the microservices' benefits. The litmus test must cover all the microservices' benefits we discussed earlier in this chapter. For a given use case, if there are no quantifiable benefits or the cost outweighs the benefit then the use case may not be the right choice for microservices.

Let's discuss some commonly used scenarios that are suitable candidates for a microservices architecture:

- Migrating a monolithic application due to improvements required in scalability, manageability, agility, or speed of delivery. Another similar scenario is rewriting an end-of-life heavily used legacy application. In both cases, microservices present an opportunity. Using a microservices architecture, it is possible to replatform a legacy application by slowly transforming functions to microservices. There are benefits in this approach. There is no humongous upfront investment required, no major disruption to business, and no severe business risks. As the service dependencies are known, the microservices dependencies can be well managed.
- Utility computing scenarios such as integrating an optimization service, forecasting service, price calculation service, prediction service, offer service, recommendation service, and so on are good candidates for microservices. These are independent stateless computing units that accept certain data, apply algorithms, and return the results. Independent technical services such as the communication service, the encryption service, authentication services, and so on are also good candidates for microservices.

- In many cases, we can build headless business applications or services that are autonomous in nature – for instance, the payment service, login service, flight search service, customer profile service, notification service, and so on. These are normally reused across multiple channels and, hence, are good candidates for building them as microservices.
- There could be micro or macro applications that serve a single purpose and performing a single responsibility. A simple time tracking application is an example of this category. All it does is capture the time, duration, and task performed. Common-use enterprise applications are also candidates for microservices.
- Backend services of a well-architected, responsive client-side MVC web application (the **Backend as a Service (BaaS)** scenario) load data on demand in response to the user navigation. In most of these scenarios, data could be coming from multiple logically different data sources as described in the *Fly By Points* example mentioned earlier.
- Highly agile applications, applications demanding speed of delivery or time to market, innovation pilots, applications selected for DevOps, applications of the System of Innovation type, and so on could also be considered as potential candidates for the microservices architecture.
- Applications that we could anticipate getting benefits from microservices such as polyglot requirements, applications that require **Command Query Responsibility segregations (CQRS)**, and so on are also potential candidates of the microservices architecture.

If the use case falls into any of these categories, it is a potential candidate for the microservices architecture.

There are few scenarios in which we should consider avoiding microservices:

- If the organization's policies are forced to use centrally managed heavyweight components such as ESB to host a business logic or if the organization has any other policies that hinder the fundamental principles of microservices, then microservices are not the right solution unless the organizational process is relaxed.
- If the organization's culture, processes, and so on are based on the traditional waterfall delivery model, lengthy release cycles, matrix teams, manual deployments and cumbersome release processes, no infrastructure provisioning, and so on, then microservices may not be the right fit. This is underpinned by Conway's Law. This states that there is a strong link between the organizational structure and software it creates.



Read more about the Conway's Law at:

http://www.melconway.com/Home/Conways_Law.html

Microservices early adopters

Many organizations have already successfully embarked on their journey to the microservices world. In this section, we will examine some of the frontrunners on the microservices space to analyze why they did what they did and how they did it. We will conduct some analysis at the end to draw some conclusions:

- **Netflix** (www.netflix.com): Netflix, an international on-demand media streaming company, is a pioneer in the microservices space. Netflix transformed their large pool of developers developing traditional monolithic code to smaller development teams producing microservices. These microservices work together to stream digital media to millions of Netflix customers. At Netflix, engineers started with monolithic, went through the pain, and then broke the application into smaller units that are loosely coupled and aligned to the business capability.
- **Uber** (www.uber.com): Uber, an international transportation network company, began in 2008 with a monolithic architecture with a single code base. All services were embedded into the monolithic application. When Uber expanded their business from one city to multiple cities, the challenges started. Uber then moved to SOA-based architecture by breaking the system into smaller independent units. Each module was given to different teams and empowered them to choose their language, framework, and database. Uber has many microservices deployed in their ecosystem using RPC and REST.
- **Airbnb** (www.airbnb.com): Airbnb, a world leader providing a trusted marketplace for accommodation, started with a monolithic application that performed all the required functions of the business. Airbnb faced scalability issues with increased traffic. A single code base became too complicated to manage, resulted in a poor separation of concerns, and ran into performance issues. Airbnb broke their monolithic application into smaller pieces with separate code bases running on separate machines with separate deployment cycles. Airbnb developed their own microservices or SOA ecosystem around these services.

- **Orbitz** (www.orbitz.com): Orbitz, an online travel portal, started with a monolithic architecture in the 2000s with a web layer, a business layer, and a database layer. As Orbitz expanded their business, they faced manageability and scalability issues with monolithic-tiered architecture. Orbitz then went through continuous architecture changes. Later, Orbitz broke down their monolithic to many smaller applications.
- **eBay** (www.ebay.com): eBay, one of the largest online retailers, started in the late 1990s with a monolithic Perl application and FreeBSD as the database. eBay went through scaling issues as the business grew. It was consistently investing in improving its architecture. In the mid 2000s, eBay moved to smaller decomposed systems based on Java and web services. They employed database partitions and functional segregation to meet the required scalability.
- **Amazon** (www.amazon.com): Amazon, one of the largest online retailer websites, was run on a big monolithic application written on C++ in 2001. The well-architected monolithic application was based on a tiered architecture with many modular components. However, all these components were tightly coupled. As a result, Amazon was not able to speed up their development cycle by splitting teams into smaller groups. Amazon then separated out the code as independent functional services, wrapped with web services, and eventually advanced to microservices.
- **Gilt** (www.gilt.com): Gilt, an online shopping website, began in 2007 with a tiered monolithic Rails application and a Postgres database at the back. Similarly to many other applications, as traffic volumes increased, the web application was not able to provide the required resiliency. Gilt went through an architecture overhaul by introducing Java and polyglot persistence. Later, Gilt moved to many smaller applications using the microservices concept.
- **Twitter** (www.twitter.com): Twitter, one of the largest social websites, began with a three-tiered monolithic rails application in the mid 2000s. Later, when Twitter experienced growth in its user base, they went through an architecture-refactoring cycle. With this refactoring, Twitter moved away from a typical web application to an API-based even driven core. Twitter uses Scala and Java to develop microservices with polyglot persistence.
- **Nike** (www.nike.com): Nike, the world leader in apparel and footwear, transformed their monolithic applications to microservices. Similarly to many other organizations, Nike too was run with age-old legacy applications that were hardly stable. In their journey, Nike moved to heavyweight commercial products with an objective to stabilize legacy applications but ended up in monolithic applications that were expensive to scale, had long release cycles, and needed too much manual work to deploy and manage applications. Later, Nike moved to a microservices-based architecture that brought down the development cycle considerably.

The common theme is monolithic migrations

When we analyze the preceding enterprises, there is one common theme. All these enterprises started with monolithic applications and transitioned to a microservices architecture by applying learning and pain points from their previous editions.

Even today, many start-ups begin with monolith as it is easy to start, conceptualize, and then slowly move to microservices when the demand arises. Monolithic to microservices migration scenarios have an added advantage: they have all the information upfront, readily available for refactoring.

Though, for all these enterprises, it is monolithic transformation, the catalysts were different for different organizations. Some of the common motivations are a lack of scalability, long development cycles, process automation, manageability, and changes in the business models.

While monolithic migrations are no-brainers, there are opportunities to build microservices from the ground up. More than building ground-up systems, look for opportunities to build smaller services that are quick wins for business—for example, adding a trucking service to an airline's end-to-end cargo management system or adding a customer scoring service to a retailer's loyalty system. These could be implemented as independent microservices exchanging messages with their respective monolithic applications.

Another point is that many organizations use microservices only for their business-critical customer engagement applications, leaving the rest of the legacy monolithic applications to take their own trajectory.

Another important observation is that most of the organizations examined previously are at different levels of maturity in their microservices journey. When eBay transitioned from a monolithic application in the early 2000s, they functionally split the application into smaller, independent, and deployable units. These logically divided units are wrapped with web services. While single responsibility and autonomy are their underpinning principles, the architectures are limited to the technologies and tools available at that point in time. Organizations such as Netflix and Airbnb built capabilities of their own to solve the specific challenges they faced. To summarize, all of these are not truly microservices, but are small, business-aligned services following the same characteristics.

There is no state called "definite or ultimate microservices". It is a journey and is evolving and maturing day by day. The mantra for architects and developers is the replaceability principle; build an architecture that maximizes the ability to replace its parts and minimizes the cost of replacing its parts. The bottom line is that enterprises shouldn't attempt to develop microservices by just following the hype.

Summary

In this chapter, you learned about the fundamentals of microservices with the help of a few examples.

We explored the evolution of microservices from traditional monolithic applications. We examined some of the principles and the mind shift required for modern application architectures. We also took a look at the characteristics and benefit of microservices and use cases. In this chapter, we established the microservices' relationship with service-oriented architecture and Twelve-Factor Apps. Lastly, we analyzed examples of a few enterprises from different industries.

We will develop a few sample microservices in the next chapter to bring more clarity to our learnings in this chapter.

2

Building Microservices with Spring Boot

Developing microservices is not so tedious anymore thanks to the powerful Spring Boot framework. Spring Boot is a framework to develop production-ready microservices in Java.

This chapter will move from the microservices theory explained in the previous chapter to hands-on practice by reviewing code samples. This chapter will introduce the Spring Boot framework and explain how Spring Boot can help build RESTful microservices in line with the principles and characteristics discussed in the previous chapter. Finally, some of the features offered by Spring Boot to make microservices production-ready will be reviewed.

By the end of this chapter, you will have learned about:

- Setting up the latest Spring development environment
- Developing RESTful services using the Spring framework
- Using Spring Boot to build fully qualified microservices
- Useful Spring Boot features to build production-ready microservices

Setting up a development environment

To crystalize microservices concepts, a couple of microservices will be built. For this, it is assumed that the following components are installed:

- **JDK 1.8:** <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- **Spring Tool Suite 3.7.2 (STS):** <https://spring.io/tools/sts/all>
- **Maven 3.3.1:** <https://maven.apache.org/download.cgi>

Alternately, other IDEs such as IntelliJ IDEA, NetBeans, or Eclipse could be used. Similarly, alternate build tools such as Gradle can be used. It is assumed that the Maven repository, class path, and other path variables are set properly to run STS and Maven projects.

This chapter is based on the following versions of Spring libraries:

- Spring Framework 4.2.6.RELEASE
- Spring Boot 1.3.5.RELEASE



Detailed steps to download the code bundle are mentioned in the Preface of this book. Have a look.

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Spring-Microservices>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Developing a RESTful service – the legacy approach

This example will review the traditional RESTful service development before jumping deep into Spring Boot.

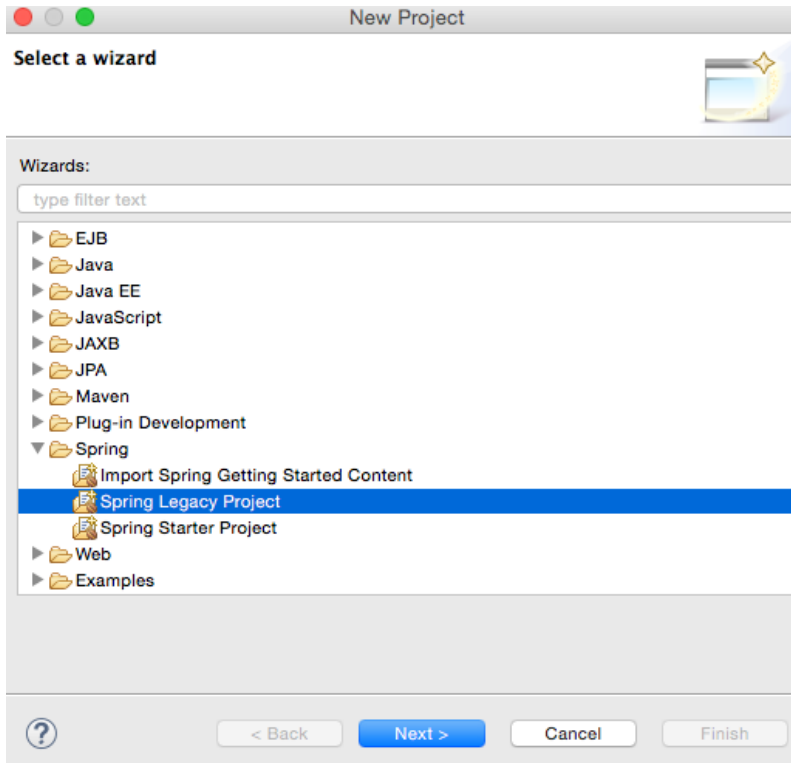
STS will be used to develop this REST/JSON service.



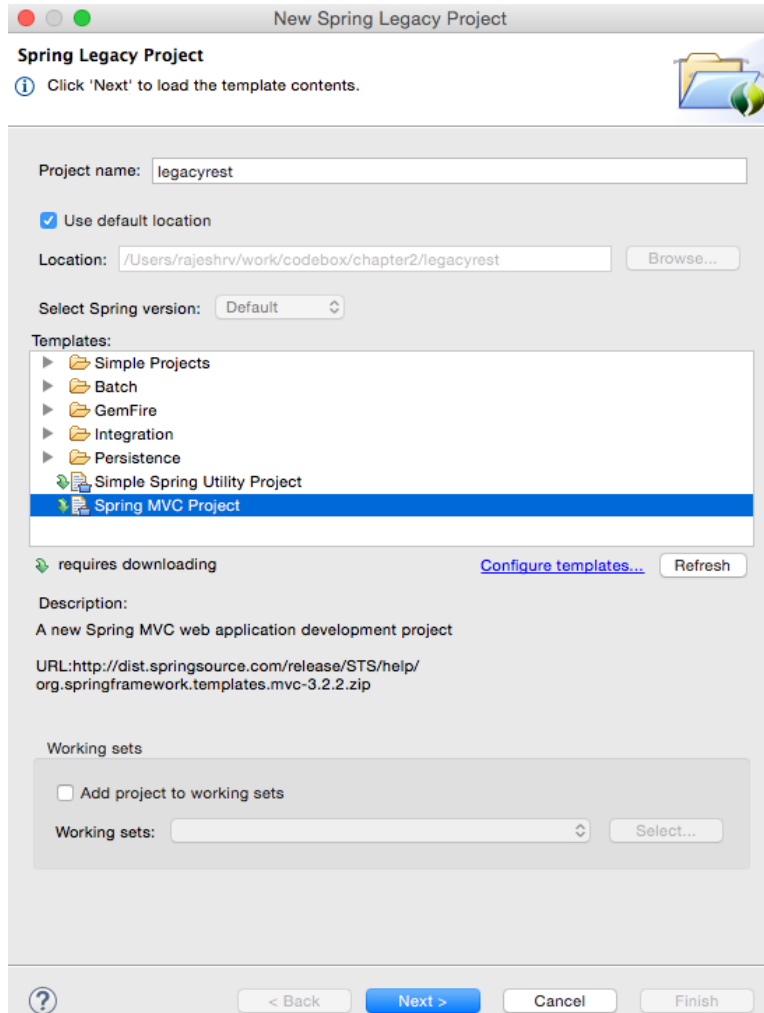
The full source code of this example is available as the `legacyrest` project in the code files of this book

The following are the steps to develop the first RESTful service

1. Start STS and set a workspace of choice for this project.
2. Navigate to **File | New | Project**.
3. Select **Spring Legacy Project** as shown in the following screenshot and click on **Next**:



4. Select **Spring MVC Project** as shown in the following diagram and click on **Next**:



5. Select a top-level package name of choice. This example uses `org.rvslab.chapter2.legacyrest` as the top-level package.
6. Then, click on **Finish**.
7. This will create a project in the STS workspace with the name `legacyrest`. Before proceeding further, `pom.xml` needs editing.

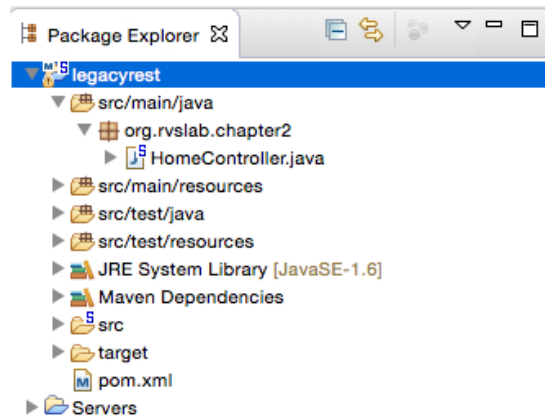
8. Change the Spring version to 4.2.6.RELEASE, as follows:

```
<org.springframework-version>4.2.6.RELEASE</org.springframework-version>
```

9. Add **Jackson** dependencies in the `pom.xml` file for JSON-to-POJO and POJO-to-JSON conversions. Note that the `2.*.*` version is used to ensure compatibility with Spring 4.

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.6.4</version>
</dependency>
```

10. Some Java code needs to be added. In **Java Resources**, under **legacyrest**, expand the package and open the default **HomeController.java** file



11. The default implementation is targeted more towards the MVC project. Rewriting `HomeController.java` to return a JSON value in response to the REST call will do the trick. The resulting `HomeController.java` file will look similar to the following:

```
@RestController
public class HomeController {
    @RequestMapping("/")
    public Greet sayHello() {
        return new Greet("Hello World!");
    }
}
```

```
class Greet {  
    private String message;  
    public Greet(String message) {  
        this.message = message;  
    }  
    //add getter and setter  
}
```

Examining the code, there are now two classes:

- Greet: This is a simple Java class with getters and setters to represent a data object. There is only one attribute in the Greet class, which is message.
- HomeController.java: This is nothing but a Spring controller REST endpoint to handle HTTP requests.

Note that the annotation used in HomeController is @RestController, which automatically injects @Controller and @ResponseBody and has the same effect as the following code:

```
@Controller  
@ResponseBody  
public class HomeController { }
```

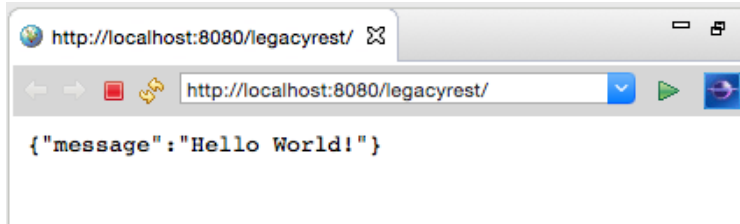
12. The project can now be run by right-clicking on **legacyrest**, navigating to **Run As | Run On Server**, and then selecting the default server (**Pivotal tc Server Developer Edition v3.1**) that comes along with STS.

This should automatically start the server and deploy the web application on the TC server.

If the server started properly, the following message will appear in the console:

```
INFO : org.springframework.web.servlet.DispatcherServlet -  
FrameworkServlet 'appServlet': initialization completed in 906 ms  
May 08, 2016 8:22:48 PM org.apache.catalina.startup.Catalina start  
INFO: Server startup in 2289 ms
```

13. If everything is fine, STS will open a browser window to `http://localhost:8080/legacyrest/` and display the JSON object as shown in the browser. Right-click on and navigate to **legacyrest** | **Properties** | **Web Project Settings** and review **Context Root** to identify the context root of the web application:



The alternate build option is to use Maven. Right-click on the project and navigate to **Run As** | **Maven install**. This will generate `chapter2-1.0.0-BUILD-SNAPSHOT.war` under the target folder. This war is deployable in any servlet container such as Tomcat, JBoss, and so on.

Moving from traditional web applications to microservices

Carefully examining the preceding RESTful service will reveal whether this really constitutes a microservice. At first glance, the preceding RESTful service is a fully qualified interoperable REST/JSON service. However, it is not fully autonomous in nature. This is primarily because the service relies on an underlying application server or web container. In the preceding example, a war was explicitly created and deployed on a Tomcat server.

This is a traditional approach to developing RESTful services as a web application. However, from the microservices point of view, one needs a mechanism to develop services as executables, self-contained JAR files with an embedded HTTP listener

Spring Boot is a tool that allows easy development of such kinds of services. Dropwizard and WildFly Swarm are alternate server-less RESTful stacks.

Using Spring Boot to build RESTful microservices

Spring Boot is a utility framework from the Spring team to bootstrap Spring-based applications and microservices quickly and easily. The framework uses an opinionated approach over configurations for decision making, thereby reducing the effort required in writing a lot of boilerplate code and configurations. Using the 80-20 principle, developers should be able to kickstart a variety of Spring applications with many default values. Spring Boot further presents opportunities for the developers to customize applications by overriding the autoconfigured values

Spring Boot not only increases the speed of development but also provides a set of production-ready ops features such as health checks and metrics collection. As Spring Boot masks many configuration parameters and abstracts many lower-level implementations, it minimizes the chance of error to a certain extent. Spring Boot recognizes the nature of the application based on the libraries available in the class path and runs the autoconfiguration classes packaged in these libraries

Often, many developers mistakenly see Spring Boot as a code generator, but in reality, it is not. Spring Boot only autoconfigures build files—for example, POM file in the case of Maven. It also sets properties, such as data source properties, based on certain opinionated defaults. Take a look at the following code:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
</dependency>
```

For instance, in the preceding case, Spring Boot understands that the project is set to use the Spring Data JPA and HSQL databases. It automatically configures the driver class and other connection parameters.

One of the great outcomes of Spring Boot is that it almost eliminates the need to have traditional XML configurations. Spring Boot also enables microservices' development by packaging all the required runtime dependencies in a fat executable JAR file

Getting started with Spring Boot

There are different ways that Spring Boot-based application development can be started:

- Using the Spring Boot CLI as a command-line tool
- Using IDEs such as STS to provide Spring Boot, which are supported out of the box
- Using the Spring Initializr project at <http://start.spring.io>

All these three options will be explored in this chapter, developing a variety of sample services.

Developing the Spring Boot microservice using the CLI

The easiest way to develop and demonstrate Spring Boot's capabilities is using the Spring Boot CLI, a command-line tool. Perform the following steps:

1. Install the Spring Boot command-line tool by downloading the `spring-boot-cli-1.3.5.RELEASE-bin.zip` file from <http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/1.3.5.RELEASE/spring-boot-cli-1.3.5.RELEASE-bin.zip>.
2. Unzip the file into a directory of your choice. Open a terminal window and change the terminal prompt to the `bin` folder.

Ensure that the `bin` folder is added to the system path so that Spring Boot can be run from any location.

3. Verify the installation with the following command. If successful, the Spring CLI version will be printed in the console:

```
$spring --version  
Spring CLI v1.3.5.RELEASE
```

4. As the next step, a quick REST service will be developed in Groovy, which is supported out of the box in Spring Boot. To do so, copy and paste the following code using any editor of choice and save it as `myfirstapp.groovy` in any folder:

```
@RestController
class HelloWorldController {
    @RequestMapping("/")
    String sayHello() {
        "Hello World!"
    }
}
```

5. In order to run this Groovy application, go to the folder where `myfirstapp.groovy` is saved and execute the following command. The last few lines of the server start-up log will be similar to the following:

```
$spring run myfirstapp.groovy
```

```
2016-05-09 18:13:55.351 INFO 35861 --- [nio-8080-exec-1]
o.s.web.servlet.DispatcherServlet : FrameworkServlet
'dispatcherServlet': initialization started

2016-05-09 18:13:55.375 INFO 35861 --- [nio-8080-exec-1]
o.s.web.servlet.DispatcherServlet : FrameworkServlet
'dispatcherServlet': initialization completed in 24 ms
```

6. Open a browser window and go to `http://localhost:8080`; the browser will display the following message:

Hello World!

There is no war file created, and no Tomcat server was run. Spring Boot automatically picked up Tomcat as the webserver and embedded it into the application. This is a very basic, minimal microservice. The `@RestController` annotation, used in the previous code, will be examined in detail in the next example.

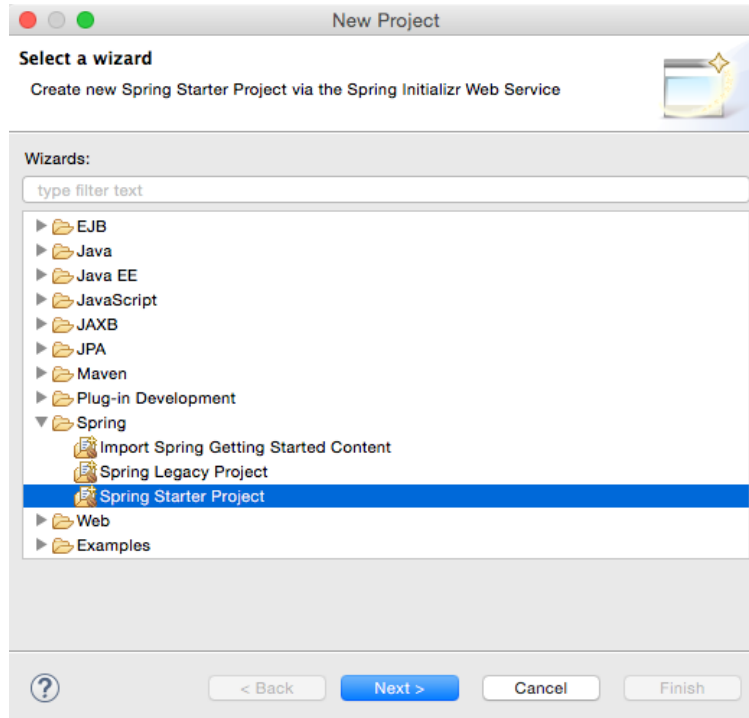
Developing the Spring Boot Java microservice using STS

In this section, developing another Java-based REST/JSON Spring Boot service using STS will be demonstrated.



The full source code of this example is available as the `chapter2.bootrest` project in the code files of this book

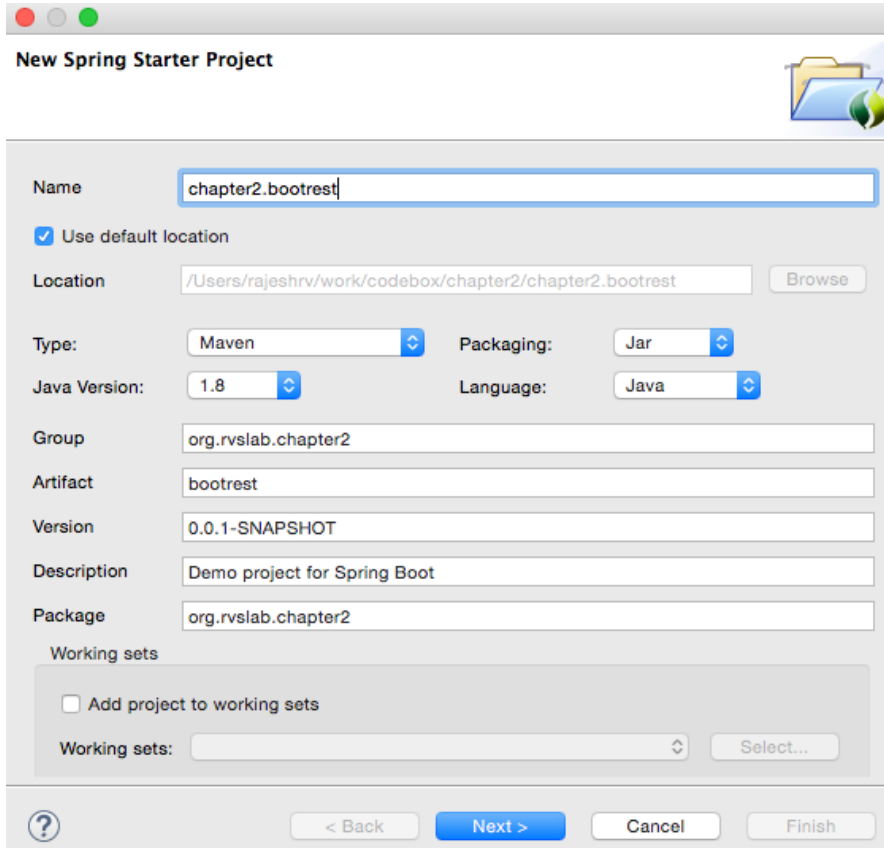
1. Open STS, right-click within the **Project Explorer** window, navigate to **New | Project**, and select **Spring Starter Project**, as shown in the following screenshot, and click on **Next**:



Spring Starter Project is a basic template wizard that provides a number of other starter libraries to select from.

2. Type the project name as `chapter2.bootrest` or any other name of your choice. It is important to choose the packaging as JAR. In traditional web applications, a war file is created and then deployed to a servlet container, whereas Spring Boot packages all the dependencies to a self-contained, autonomous JAR file with an embedded HTTP listener

3. Select 1.8 under **Java Version**. Java 1.8 is recommended for Spring 4 applications. Change the other Maven properties such as **Group**, **Artifact**, and **Package**, as shown in the following screenshot:



New Spring Starter Project

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

Package:

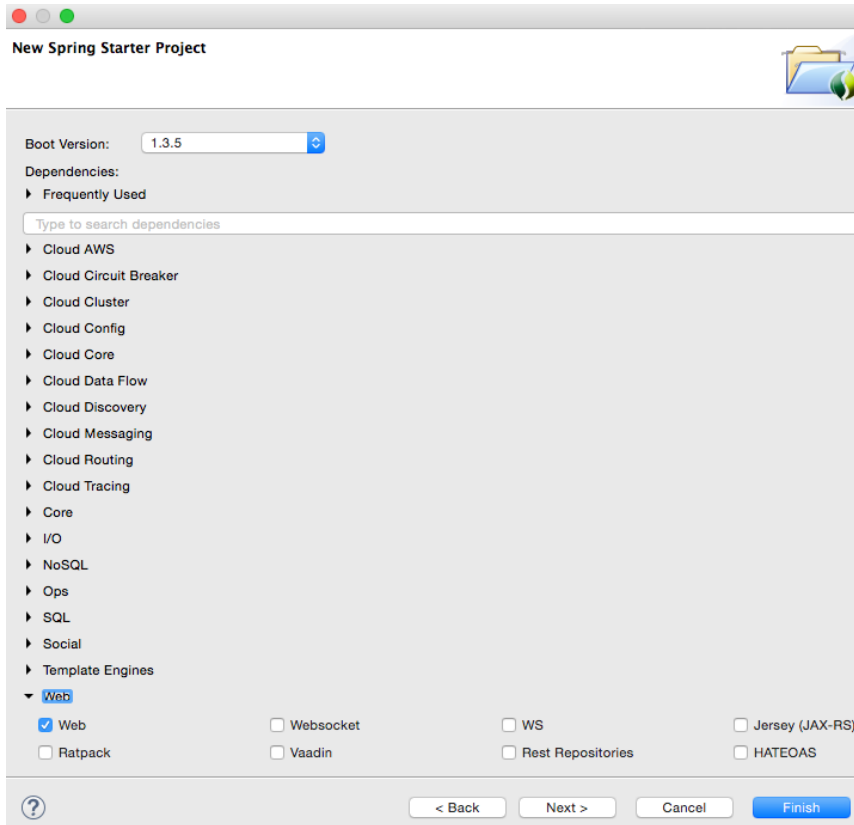
Working sets

☐ Add project to working sets

Working sets:

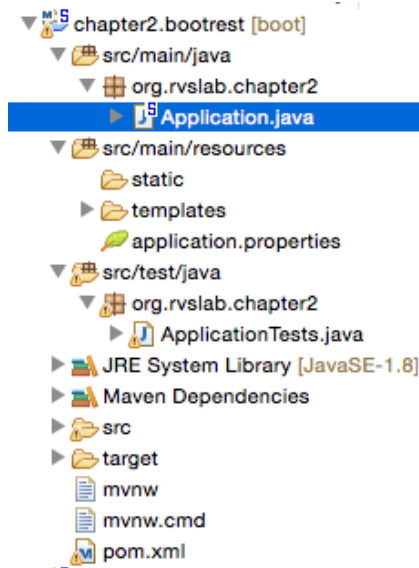
4. Once completed, click on **Next**.

- The wizard will show the library options. In this case, as the REST service is developed, select **Web** under **Web**. This is an interesting step that tells Spring Boot that a Spring MVC web application is being developed so that Spring Boot can include the necessary libraries, including Tomcat as the HTTP listener and other configurations, as required



6. Click on **Finish**.

This will generate a project named `chapter2.bootrest` in **Project Explorer** in STS:



7. Take a moment to examine the generated application. Files that are of interest are:

- `pom.xml`
- `Application.java`
- `Application.properties`
- `ApplicationTests.java`

Examining the POM file

The parent element is one of the interesting aspects in the `pom.xml` file. Take a look at the following:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.4.RELEASE</version>
</parent>
```

The `spring-boot-starter-parent` pattern is a **bill of materials (BOM)**, a pattern used by Maven's dependency management. BOM is a special kind of POM file used to manage different library versions required for a project. The advantage of using the `spring-boot-starter-parent` POM file is that developers need not worry about finding the right compatible versions of different libraries such as Spring, Jersey, JUnit, Logback, Hibernate, Jackson, and so on. For instance, in our first legacy example, a specific version of the Jackson library was added to work with Spring 4. In this example, these are taken care of by the `spring-boot-starter-parent` pattern.

The starter POM file has a list of Boot dependencies, sensible resource filtering, and sensible plug-in configurations required for the Maven builds



Refer to <https://github.com/spring-projects/spring-boot/blob/1.3.x/spring-boot-dependencies/pom.xml> to take a look at the different dependencies provided in the starter parent (version 1.3.x). All these dependencies can be overridden if required.

The starter POM file itself does not add JAR dependencies to the project. Instead, it will only add library versions. Subsequently, when dependencies are added to the POM file, they refer to the library versions from this POM file. A snapshot of some of the properties are as shown as follows:

```
<spring-boot.version>1.3.5.BUILD-SNAPSHOT</spring-boot.version>
<hibernate.version>4.3.11.Final</hibernate.version>
<jackson.version>2.6.6</jackson.version>
<jersey.version>2.22.2</jersey.version>
<logback.version>1.1.7</logback.version>
<spring.version>4.2.6.RELEASE</spring.version>
<spring-data-releasetrain.version>Gosling-SR4</spring-data-releasetrain.version>
<tomcat.version>8.0.33</tomcat.version>
```

Reviewing the dependency section, one can see that this is a clean and neat POM file with only two dependencies, as follows:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
```



```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
</dependencies>
```

As `web` is selected, `spring-boot-starter-web` adds all dependencies required for a Spring MVC project. It also includes dependencies to Tomcat as an embedded HTTP listener. This provides an effective way to get all the dependencies required as a single bundle. Individual dependencies could be replaced with other libraries, for example replacing Tomcat with Jetty.

Similar to `web`, Spring Boot comes up with a number of `spring-boot-starter-*` libraries, such as `amqp`, `aop`, `batch`, `data-jpa`, `thymeleaf`, and so on.

The last thing to be reviewed in the `pom.xml` file is the Java 8 property. By default, the parent POM file adds Java 6. It is recommended to override the Java version to 8 for Spring:

```
<java.version>1.8</java.version>
```

Examining Application.java

Spring Boot, by default, generated a `org.rvslab.chapter2.Application.java` class under `src/main/java` to bootstrap, as follows:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

There is only a `main` method in `Application`, which will be invoked at startup as per the Java convention. The `main` method bootstraps the Spring Boot application by calling the `run` method on `SpringApplication`. `Application.class` is passed as a parameter to tell Spring Boot that this is the primary component.

More importantly, the magic is done by the `@SpringBootApplication` annotation. The `@SpringBootApplication` annotation is a top-level annotation that encapsulates three other annotations, as shown in the following code snippet:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {
```

The `@Configuration` annotation hints that the contained class declares one or more `@Bean` definitions. The `@Configuration` annotation is meta-annotated with `@Component`; therefore, it is a candidate for component scanning.

The `@EnableAutoConfiguration` annotation tells Spring Boot to automatically configure the Spring application based on the dependencies available in the class path.

Examining application.properties

A default `application.properties` file is placed under `src/main/resources`. It is an important file to configure any required properties for the Spring Boot application. At the moment, this file is kept empty and will be revisited with some test cases later in this chapter.

Examining ApplicationTests.java

The last file to be examined is `ApplicationTests.java` under `src/test/java`. This is a placeholder to write test cases against the Spring Boot application.

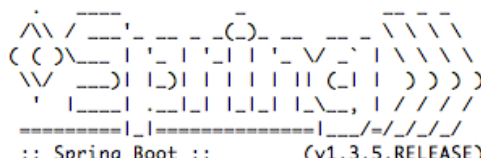
To implement the first RESTful service, add a REST endpoint, as follows

1. One can edit `Application.java` under `src/main/java` and add a RESTful service implementation. The RESTful service is exactly the same as what was done in the previous project. Append the following code at the end of the `Application.java` file

```
@RestController
class GreetingController{
    @RequestMapping("/")
    Greet greet(){
        return new Greet("Hello World!");
    }
}
```

```
}  
class Greet {  
    private String message;  
    public Greet() {}  
  
    public Greet(String message) {  
        this.message = message;  
    }  
    //add getter and setter  
}
```

2. To run, navigate to **Run As | Spring Boot App**. Tomcat will be started on the 8080 port:



```
:: Spring Boot ::      (v1.3.5.RELEASE)  
  
2016-05-11 16:49:10.236 INFO 41130 --- [          main]  
org.rvslab.chapter2.Application : Starting Application on rvslab.local with  
PID 41130 (/Users/rajeshrv/work/codebox/chapter2/chapter2.bootrest/target/classes)
```

We can notice from the log that:

- Spring Boot get its own process ID (in this case, it is 41130)
 - Spring Boot is automatically started with the Tomcat server at the localhost, port 8080.
3. Next, open a browser and point to <http://localhost:8080>. This will show the JSON response as shown in the following screenshot:



The screenshot shows a web browser window with the address bar set to <http://localhost:8080/>. The page content displays the JSON response: `{"message": "Hello World!"}`.

A key difference between the legacy service and this one is that the Spring Boot service is self-contained. To make this clearer, run the Spring Boot application outside STS. Open a terminal window, go to the project folder, and run Maven, as follows:

```
$ maven install
```

This will generate a fat JAR file under the target folder of the project. Running the application from the command line shows:

```
$java -jar target/bootrest-0.0.1-SNAPSHOT.jar
```

As one can see, `bootrest-0.0.1-SNAPSHOT.jar` is self-contained and could be run as a standalone application. At this point, the JAR is as thin as 13 MB. Even though the application is no more than just "Hello World", the Spring Boot service just developed, practically follows the principles of microservices.

Testing the Spring Boot microservice

There are multiple ways to test REST/JSON Spring Boot microservices. The easiest way is to use a web browser or a curl command pointing to the URL, as follows:

```
curl http://localhost:8080
```

There are number of tools available to test RESTful services, such as Postman, Advanced REST client, SOAP UI, Paw, and so on.

In this example, to test the service, the default test class generated by Spring Boot will be used.

Adding a new test case to `ApplicationTests.java` results in:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)
@WebIntegrationTest
public class ApplicationTests {
    @Test
    public void testVanillaService() {
        RestTemplate restTemplate = new RestTemplate();
        Greet greet = restTemplate.getForObject
            ("http://localhost:8080", Greet.class);
        Assert.assertEquals("Hello World!", greet.getMessage());
    }
}
```

Note that `@WebIntegrationTest` is added and `@WebAppConfiguration` removed at the class level. The `@WebIntegrationTest` annotation is a handy annotation that ensures that the tests are fired against a fully up-and-running server. Alternately, a combination of `@WebAppConfiguration` and `@IntegrationTest` will give the same result.

Also note that `RestTemplate` is used to call the RESTful service. `RestTemplate` is a utility class that abstracts the lower-level details of the HTTP client.

To test this, one can open a terminal window, go to the project folder, and run `mvn install`.

Developing the Spring Boot microservice using Spring Initializr – the HATEOAS example

In the next example, Spring Initializr will be used to create a Spring Boot project. Spring Initializr is a drop-in replacement for the STS project wizard and provides a web UI to configure and generate a Spring Boot project. One of the advantages of Spring Initializr is that it can generate a project through the website that then can be imported into any IDE.

In this example, the concept of **HATEOAS** (short for **H**ypertext **A**s **T**he **E**ngine **O**f **A**pplication **S**tate) for REST-based services and the **HAL** (**H**ypertext **A**pplication **L**anguage) browser will be examined.

HATEOAS is a REST service pattern in which navigation links are provided as part of the payload metadata. The client application determines the state and follows the transition URLs provided as part of the state. This methodology is particularly useful in responsive mobile and web applications in which the client downloads additional data based on user navigation patterns.

The HAL browser is a handy API browser for `hal+json` data. HAL is a format based on JSON that establishes conventions to represent hyperlinks between resources. HAL helps APIs be more explorable and discoverable.



The full source code of this example is available as the `chapter2.boothateoas` project in the code files of this book

Here are the concrete steps to develop a HATEOAS sample using Spring Initializr:

1. In order to use Spring Initializr, go to <https://start.spring.io>:

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.3.5

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies

[Generate Project](#)

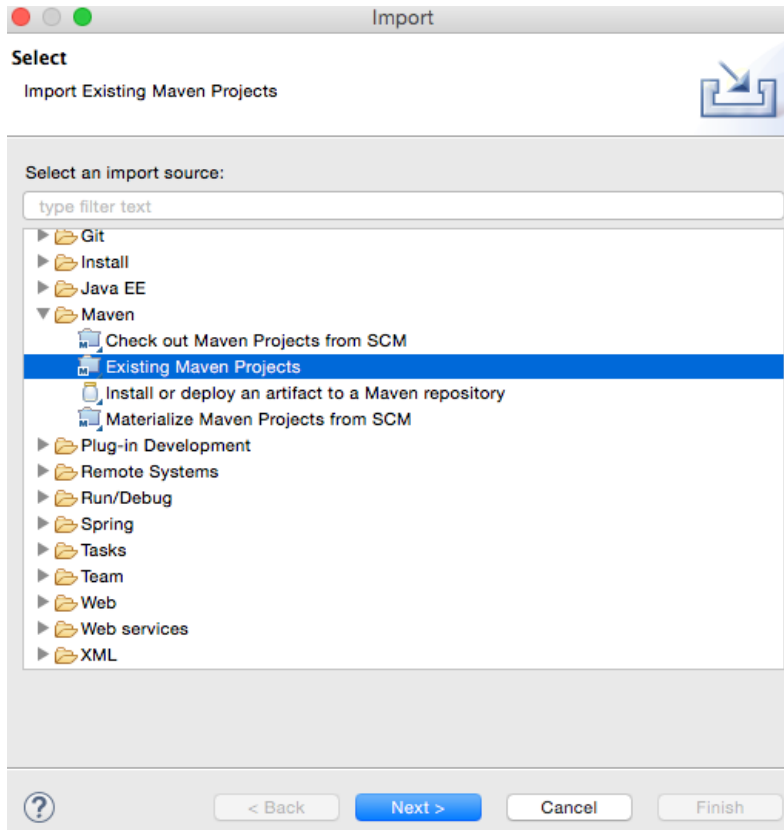
Don't know what to look for? Want more options? [Switch to the full version.](#)

2. Fill the details, such as whether it is a Maven project, Spring Boot version, group, and artifact ID, as shown earlier, and click on **Switch to the full version** link under the **Generate Project** button. Select **Web**, **HATEOAS**, and **Rest Repositories HAL Browser**. Make sure that the Java version is 8 and the package type is selected as **JAR**:

Web

- ☐ Web
Full-stack web development with Tomcat and Spring MVC
- ☐ Websocket
Websocket development with SockJS and STOMP
- ☐ WS
Contract-first SOAP service development with Spring Web Services
- ☐ Jersey (JAX-RS)
the Jersey RESTful Web Services framework
- ☐ Ratpack
Spring Boot integration for the Ratpack framework
- ☐ Vaadin
Vaadin
- ☐ Rest Repositories
Exposing Spring Data repositories over REST via spring-data-rest-webmvc
- ☐ HATEOAS
HATEOAS-based RESTful services
- ☐ Rest Repositories HAL Browser
Browsing Spring Data REST repositories with an HTML UI
- ☐ Mobile
Simplify the development of mobile web applications with spring-mobile
- ☐ REST Docs
Document RESTful services by combining hand-written and auto-generated documentation

3. Once selected, hit the **Generate Project** button. This will generate a Maven project and download the project as a ZIP file into the download directory of the browser.
4. Unzip the file and save it to a directory of your choice
5. Open STS, go to the **File** menu and click on **Import**:



6. Navigate to **Maven | Existing Maven Projects** and click on **Next**.
7. Click on **Browse** next to **Root Directory** and select the unzipped folder. Click on **Finish**. This will load the generated Maven project into STS' **Project Explorer**.

8. Edit the `Application.java` file to add a new REST endpoint, as follows

```
@RequestMapping("/greeting")
@ResponseBody
public HttpEntity<Greet> greeting(@RequestParam(value = "name",
required = false, defaultValue = "HATEOAS") String name) {
    Greet greet = new Greet("Hello " + name);
    greet.add(linkTo(methodOn(GreetingController.
        class).greeting(name)).withSelfRel());

    return new ResponseEntity<Greet>(greet,
        HttpStatus.OK);
}
```

9. Note that this is the same `GreetingController` class as in the previous example. However, a method was added this time named `greeting`. In this new method, an additional optional request parameter is defined and defaulted to `HATEOAS`. The following code adds a link to the resulting JSON code. In this case, it adds the link to the same API:

```
greet.add(linkTo(methodOn(GreetingController.class).
greeting(name)).withSelfRel());
```

In order to do this, we need to extend the `Greet` class from `ResourceSupport`, as shown here. The rest of the code remains the same:

```
class Greet extends ResourceSupport{
```

10. The `add` method is a method in `ResourceSupport`. The `linkTo` and `methodOn` methods are static methods of `ControllerLinkBuilder`, a utility class for creating links on controller classes. The `methodOn` method will do a dummy method invocation, and `linkTo` will create a link to the controller class. In this case, we will use `withSelfRel` to point it to itself.
11. This will essentially produce a link, `/greeting?name=HATEOAS`, by default. A client can read the link and initiate another call.
12. Run this as a Spring Boot app. Once the server startup is complete, point the browser to `http://localhost:8080`.

13. This will open the HAL browser window. In the **Explorer** field, type `/greeting?name=World!` and click on the **Go** button. If everything is fine, the HAL browser will show the response details as shown in the following screenshot:

The screenshot shows the HAL Browser interface. At the top, there are navigation links: "The HAL Browser", "Go To Entry Point", and "About The HAL Browser". The main interface is divided into two columns: "Explorer" on the left and "Inspector" on the right.

Explorer: The "URL" field contains `/greeting?name=World!` and the "Go!" button is visible. Below it, the "Custom Request Headers" section is empty. The "Properties" section shows a JSON object: `{ "message": "Hello World!" }`. The "Links" section shows a table with columns: rel, title, name / index, docs, GET, and NON-GET. There is one entry for "self" with a green box next to it, indicating it is a navigable link.

Inspector: The "Response Headers" section shows a 200 OK status and headers: Date: Sat, 12 Dec 2015 18:12:43 GMT, Server: Apache-Coyote/1.1, Transfer-Encoding: Identity, Content-Type: application/hal+json; charset=UTF-8. The "Response Body" section shows a JSON object: `{ "message": "Hello World!", "_links": { "self": { "href": "http://localhost:8080/greeting?name=World!" } } }`.

As shown in the screenshot, the **Response Body** section has the result with a link with `href` pointing back to the same service. This is because we pointed the reference to itself. Also, review the **Links** section. The little green box against **self** is the navigable link.

It does not make much sense in this simple example, but this could be handy in larger applications with many related entities. Using the links provided, the client can easily navigate back and forth between these entities with ease.

What's next?

A number of basic Spring Boot examples have been reviewed so far. The rest of this chapter will examine some of the Spring Boot features that are important from a microservices development perspective. In the upcoming sections, we will take a look at how to work with dynamically configurable properties, change the default embedded web server, add security to the microservices, and implement cross-origin behavior when dealing with microservices.



The full source code of this example is available as the `chapter2.boot-advanced` project in the code files of this book

The Spring Boot configuration

In this section, the focus will be on the configuration aspects of Spring Boot. The `chapter2.bootrest` project, already developed, will be modified in this section to showcase configuration capabilities. Copy and paste `chapter2.bootrest` and rename the project as `chapter2.boot-advanced`.

Understanding the Spring Boot autoconfiguration

Spring Boot uses convention over configuration by scanning the dependent libraries available in the class path. For each `spring-boot-starter-*` dependency in the POM file, Spring Boot executes a default `AutoConfiguration` class. `AutoConfiguration` classes use the `*AutoConfiguration` lexical pattern, where `*` represents the library. For example, the autoconfiguration of JPA repositories is done through `JpaRepositoriesAutoConfiguration`.

Run the application with `--debug` to see the autoconfiguration report. The following command shows the autoconfiguration report for the `chapter2.boot-advanced` project:

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --debug
```

Here are some examples of the autoconfiguration classes

- `ServerPropertiesAutoConfiguration`
- `RepositoryRestMvcAutoConfiguration`
- `JpaRepositoriesAutoConfiguration`
- `JmsAutoConfiguration`

It is possible to exclude the autoconfiguration of certain libraries if the application has special requirements and you want to get full control of the configurations. The following is an example of excluding `DataSourceAutoConfiguration`:

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```

```
server.port 9090

spring.j
spring.jackson.date-format : String
spring.jackson.deserialization : Map<com.fasterxml.jackson.databind.DeserializationFeature, Boolean>
spring.jackson.generator : Map<com.fasterxml.jackson.core.JsonGenerator.Feature[AUTO_CLOSE, Boolean>
spring.jackson.joda-date-time-format : String
spring.jackson.locale : Locale
spring.jackson.mapper : Map<com.fasterxml.jackson.databind.MapperFeature[USE_ANNOTATIONS, Boolean>
spring.jackson.parser : Map<com.fasterxml.jackson.core.JsonParser.Feature[AUTO_CLOSE, Boolean>
spring.jackson.property-naming-strategy : String
spring.jackson.serialization : Map<com.fasterxml.jackson.databind.SerializationFeature[WRITE_DATES_WITH_TIMELINE, Boolean>
spring.jackson.serialization-inclusion : com.fasterxml.jackson.annotation.JsonInclude$Include
spring.jackson.time-zone : TimeZone
spring.jersey.application-path : String
spring.jersey.filter.order : int
spring.jersey.init : Map<String, String>
spring.jersey.type : org.springframework.boot.autoconfigure.jersey.JerseyProperties$Type
spring.jms.indi-name : String
```

Changing the location of the configuration file

```
spring.config.name= # config file name
spring.config.location= # location of config file
```

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --spring.config.name=bootrest.properties
```

Reading custom properties

At startup, `SpringApplication` loads all the properties and adds them to the `Spring Environment` class. Add a custom property to the `application.properties` file. In this case, the custom property is named `bootrest.customproperty`. Autowire the `Spring Environment` class into the `GreetingController` class. Edit the `GreetingController` class to read the custom property from `Environment` and add a log statement to print the custom property to the console.

Perform the following steps to do this:

1. Add the following property to the `application.properties` file

```
bootrest.customproperty=hello
```

2. Then, edit the `GreetingController` class as follows:

```
@Autowired
Environment env;

Greet greet() {
    logger.info("bootrest.customproperty "+
        env.getProperty("bootrest.customproperty"));
    return new Greet("Hello World!");
}
```

3. Rerun the application. The log statement prints the custom variable in the console, as follows:

```
org.rvslab.chapter2.GreetingController    : bootrest.customproperty
hello
```

Using a .yaml file for configuration

As an alternate to `application.properties`, one may use a `.yaml` file. YAML provides a JSON-like structured configuration compared to the flat properties file.

To see this in action, simply replace `application.properties` with `application.yaml` and add the following property:

```
server
  port: 9080
```

Rerun the application to see the port printed in the console.

Using multiple configuration profiles

Furthermore, it is possible to have different profiles such as development, testing, staging, production, and so on. These are logical names. Using these, one can configure different values for the same properties for different environments. This is quite handy when running the Spring Boot application against different environments. In such cases, there is no rebuild required when moving from one environment to another.

Update the `.yaml` file as follows. The Spring Boot group profiles properties based on the dotted separator:

```
spring:
  profiles: development
server:
  port: 9090
---
spring:
  profiles: production
server:
  port: 8080
```

Run the Spring Boot application as follows to see the use of profiles

```
mvn -Dspring.profiles.active=production install
mvn -Dspring.profiles.active=development install
```

Active profiles can be specified programmatically using the `@ActiveProfiles` annotation, which is especially useful when running test cases, as follows:

```
@ActiveProfiles("test")
```

Other options to read properties

The properties can be loaded in a number of ways, such as the following:

- Command-line parameters (`-Dhost.port =9090`)
- Operating system environment variables
- JNDI (`java:comp/env`)

Changing the default embedded web server

Embedded HTTP listeners can easily be customized as follows. By default, Spring Boot supports Tomcat, Jetty, and Undertow. In the following example, Tomcat is replaced with Undertow:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

Implementing Spring Boot security

It is important to secure microservices. In this section, some basic measures to secure Spring Boot microservices will be reviewed using `chapter2.bootrest` to demonstrate the security features.

Securing microservices with basic security

Adding basic authentication to Spring Boot is pretty simple. Add the following dependency to `pom.xml`. This will include the necessary Spring security library files

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Open `Application.java` and add `@EnableGlobalMethodSecurity` to the `Application` class. This annotation will enable method-level security:

```
@EnableGlobalMethodSecurity
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

The default basic authentication assumes the user as being `user`. The default password will be printed in the console at startup. Alternately, the username and password can be added in `application.properties`, as shown here:

```
security.user.name=guest
security.user.password=guest123
```

Add a new test case in `ApplicationTests` to test the secure service results, as in the following:

```
@Test
public void testSecureService() {
    String plainCreds = "guest:guest123";
    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "Basic " + new String(Base64.
encode(plainCreds.getBytes())));
    HttpEntity<String> request = new HttpEntity<String>(headers);
    RestTemplate restTemplate = new RestTemplate();

    ResponseEntity<Greet> response = restTemplate.exchange("http://
localhost:8080", HttpMethod.GET, request, Greet.class);
    Assert.assertEquals("Hello World!", response.getBody().
getMessage());
}
```

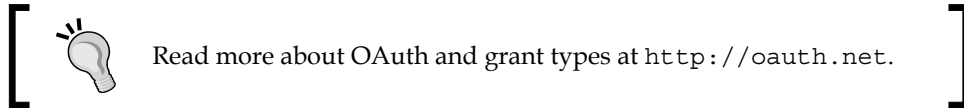
As shown in the code, a new `Authorization` request header with Base64 encoding the username-password string is created.

Rerun the application using Maven. Note that the new test case passed, but the old test case failed with an exception. The earlier test case now runs without credentials, and as a result, the server rejected the request with the following message:

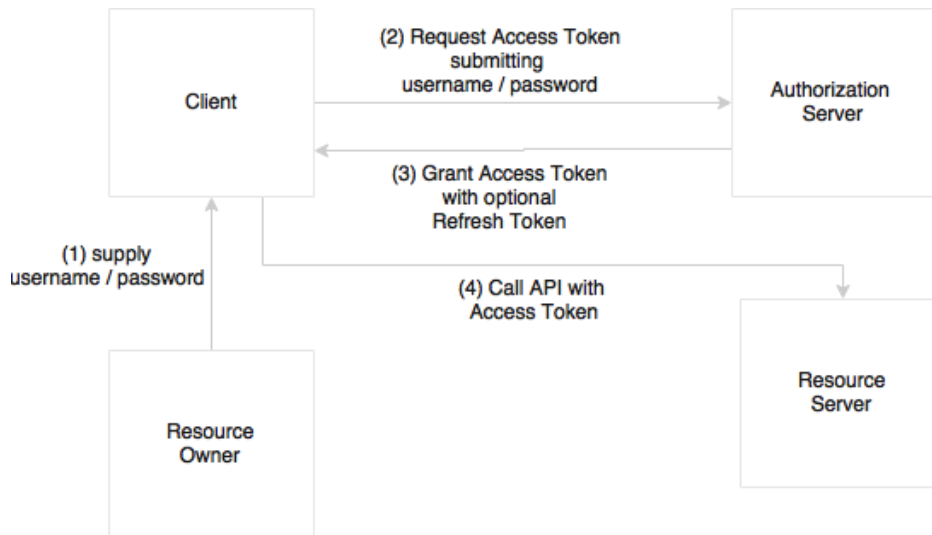
```
org.springframework.web.client.HttpClientErrorException: 401 Unauthorized
```

Securing a microservice with OAuth2

In this section, we will take a look at the basic Spring Boot configuration for OAuth2. When a client application requires access to a protected resource, the client sends a request to an authorization server. The authorization server validates the request and provides an access token. This access token is validated for every client-to-server request. The request and response sent back and forth depends on the grant type.



The resource owner password credentials grant approach will be used in this example:



In this case, as shown in the preceding diagram, the resource owner provides the client with a username and password. The client then sends a token request to the authorization server by providing the credential information. The authorization server authorizes the client and returns with an access token. On every subsequent request, the server validates the client token.

To implement OAuth2 in our example, perform the following steps:

1. As a first step, update `pom.xml` with the OAuth2 dependency, as follows:

```
<dependency>
  <groupId>org.springframework.security.oauth</groupId>
  <artifactId>spring-security-oauth2</artifactId>
  <version>2.0.9.RELEASE</version>
</dependency>
```
2. Next, add two new annotations, `@EnableAuthorizationServer` and `@EnableResourceServer`, to the `Application.java` file. The `@EnableAuthorizationServer` annotation creates an authorization server with an in-memory repository to store client tokens and provide clients with a username, password, client ID, and secret. The `@EnableResourceServer` annotation is used to access the tokens. This enables a spring security filter that is authenticated via an incoming OAuth2 token.

In our example, both the authorization server and resource server are the same. However, in practice, these two will run separately. Take a look at the following code:

```
@EnableResourceServer
@EnableAuthorizationServer
@SpringBootApplication
public class Application {
```

3. Add the following properties to the `application.properties` file

```
security.user.name=guest
security.user.password=guest123
security.oauth2.client.clientId: trustedclient
security.oauth2.client.clientSecret: trustedclient123
security.oauth2.client.authorized-grant-types: authorization_
code,refresh_token,password
security.oauth2.client.scope: openid
```

4. Then, add another test case to test OAuth2, as follows:

```

@Test
public void testOAuthService() {
    ResourceOwnerPasswordResourceDetails resource = new
ResourceOwnerPasswordResourceDetails();
    resource.setUsername("guest");
    resource.setPassword("guest123");
    resource.setAccessTokenUri("http://localhost:8080/oauth/
token");
    resource.setClientId("trustedclient");
    resource.setClientSecret("trustedclient123");
    resource.setGrantType("password");

    DefaultOAuth2ClientContext clientContext = new
DefaultOAuth2ClientContext();
    OAuth2RestTemplate restTemplate = new
OAuth2RestTemplate(resource, clientContext);

    Greet greet = restTemplate.getForObject("http://
localhost:8080", Greet.class);

    Assert.assertEquals("Hello World!", greet.getMessage());
}

```

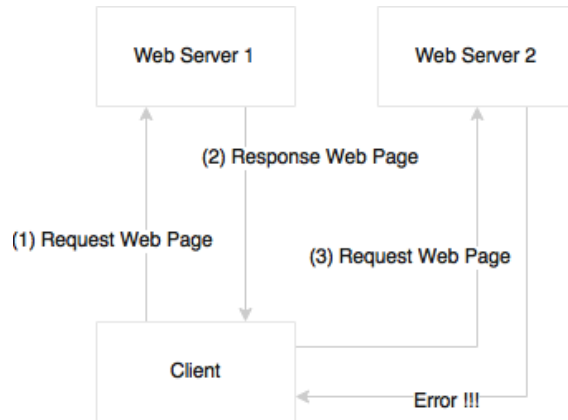
As shown in the preceding code, a special REST template, `OAuth2RestTemplate`, is created by passing the resource details encapsulated in a resource details object. This REST template handles the OAuth2 processes underneath. The access token URI is the endpoint for the token access.

5. Rerun the application using `mvn install`. The first two test cases will fail, and the new one will succeed. This is because the server only accepts OAuth2-enabled requests.

These are quick configurations provided by Spring Boot out of the box but are not good enough to be production grade. We may need to customize `ResourceServerConfigurer` and `AuthorizationServerConfigurer` to make them production-ready. This notwithstanding, the approach remains the same.

Enabling cross-origin access for microservices

Browsers are generally restricted when client-side web applications running from one origin request data from another origin. Enabling cross-origin access is generally termed as **CORS (Cross-Origin Resource Sharing)**.



This example shows how to enable cross-origin requests. With microservices, as each service runs with its own origin, it will easily get into the issue of a client-side web application consuming data from multiple origins. For instance, a scenario where a browser client accessing Customer from the Customer microservice and Order History from the Order microservices is very common in the microservices world.

Spring Boot provides a simple declarative approach to enabling cross-origin requests. The following example shows how to enable a microservice to enable cross-origin requests:

```
@RestController
class GreetingController{
    @CrossOrigin
    @RequestMapping("/")
    Greet greet(){
        return new Greet("Hello World!");
    }
}
```

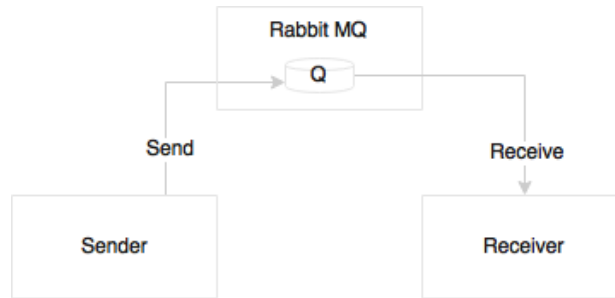
By default, all the origins and headers are accepted. We can further customize the cross-origin annotations by giving access to specific origins, as follows. The `@CrossOrigin` annotation enables a method or class to accept cross-origin requests:

```
@CrossOrigin("http://mytrustedorigin.com")
```

Global CORS can be enabled using the `WebMvcConfigurer` bean and customizing the `addCorsMappings(CorsRegistry registry)` method.

Implementing Spring Boot messaging

In an ideal case, all microservice interactions are expected to happen asynchronously using publish-subscribe semantics. Spring Boot provides a hassle-free mechanism to configure messaging solutions

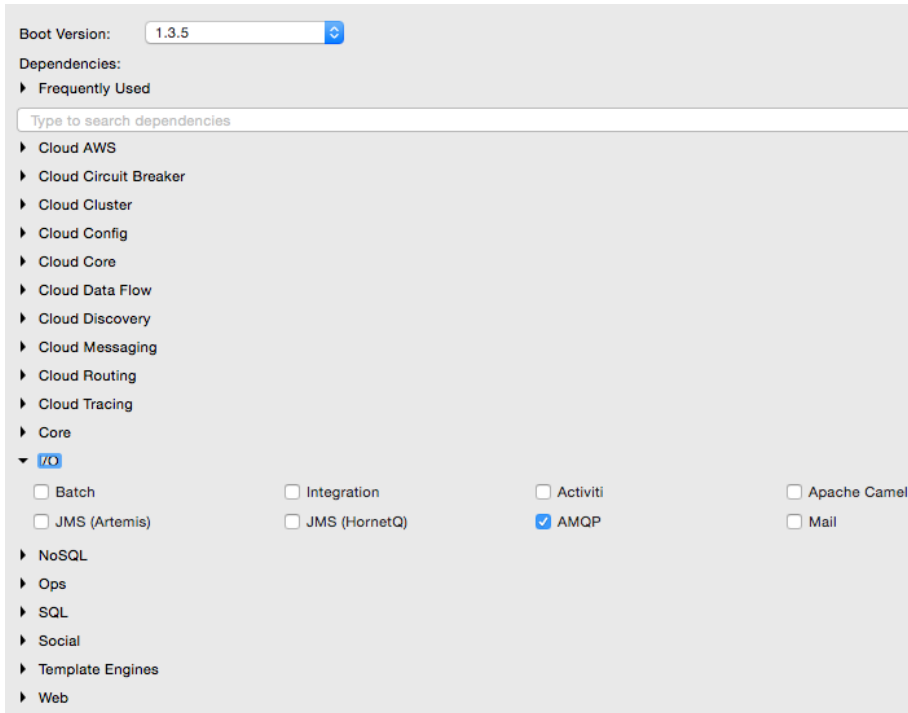


In this example, we will create a Spring Boot application with a sender and receiver, both connected through an external queue. Perform the following steps:



The full source code of this example is available as the `chapter2.bootmessaging` project in the code files of this book

1. Create a new project using STS to demonstrate this capability. In this example, instead of selecting **Web**, select **AMQP** under **I/O**:



2. Rabbit MQ will also be needed for this example. Download and install the latest version of Rabbit MQ from <https://www.rabbitmq.com/download.html>.

Rabbit MQ 3.5.6 is used in this book.

3. Follow the installation steps documented on the site. Once ready, start the RabbitMQ server via the following command:

```
$ ./rabbitmq-server
```

4. Make the configuration changes to the `application.properties` file to reflect the RabbitMQ configuration. The following configuration uses the default port, username, and password of RabbitMQ:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

5. Add a message sender component and a queue named `TestQ` of the `org.springframework.amqp.core.Queue` type to the `Application.java` file under `src/main/java`. `RabbitMessagingTemplate` is a convenient way to send messages, which will abstract all the messaging semantics. Spring Boot provides all boilerplate configurations to send messages

```
@Component
class Sender {
    @Autowired
    RabbitMessagingTemplate template;
    @Bean
    Queue queue() {
        return new Queue("TestQ", false);
    }
    public void send(String message) {
        template.convertAndSend("TestQ", message);
    }
}
```

6. To receive the message, all that needs to be used is a `@RabbitListener` annotation. Spring Boot autoconfigures all the required boilerplate configurations

```
@Component
class Receiver {
    @RabbitListener(queues = "TestQ")
    public void processMessage(String content) {
        System.out.println(content);
    }
}
```

7. The last piece of this exercise is to wire the sender to our main application and implement the `run` method of `CommandLineRunner` to initiate the message sending. When the application is initialized, it invokes the `run` method of `CommandLineRunner`, as follows:

```
@SpringBootApplication
public class Application implements CommandLineRunner{

    @Autowired
    Sender sender;

    public static void main(String[] args) {
```

```
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        sender.send("Hello Messaging...!!!");
    }
}
```

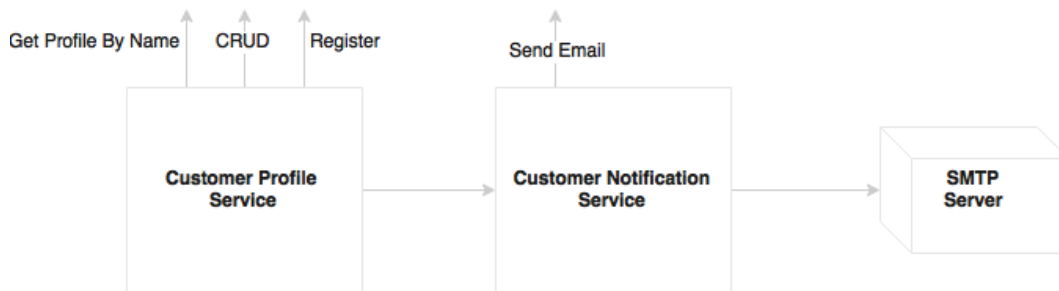
8. Run the application as a Spring Boot application and verify the output. The following message will be printed in the console:

Hello Messaging...!!!

Developing a comprehensive microservice example

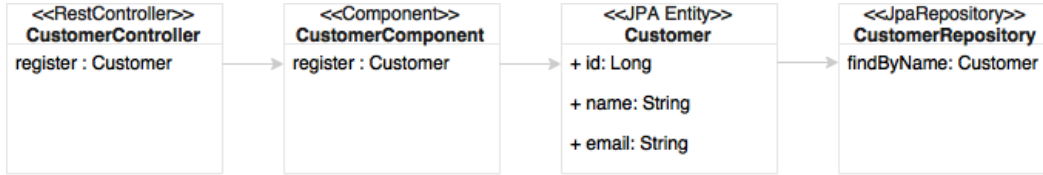
So far, the examples we have considered are no more than just a simple "Hello world." Putting together what we have learned, this section demonstrates an end-to-end Customer Profile microservice implementation. The Customer Profile microservices will demonstrate interaction between different microservices. It also demonstrates microservices with business logic and primitive data stores.

In this example, two microservices, the Customer Profile and Customer Notification services, will be developed:




As shown in the diagram, the Customer Profile microservice exposes methods to **create, read, update, and delete (CRUD)** a customer and a registration service to register a customer. The registration process applies certain business logic, saves the customer profile, and sends a message to the Customer Notification microservice. The Customer Notification microservice accepts the message sent by the registration service and sends an e-mail message to the customer using an SMTP server. Asynchronous messaging is used to integrate Customer Profile with the Customer Notification service.

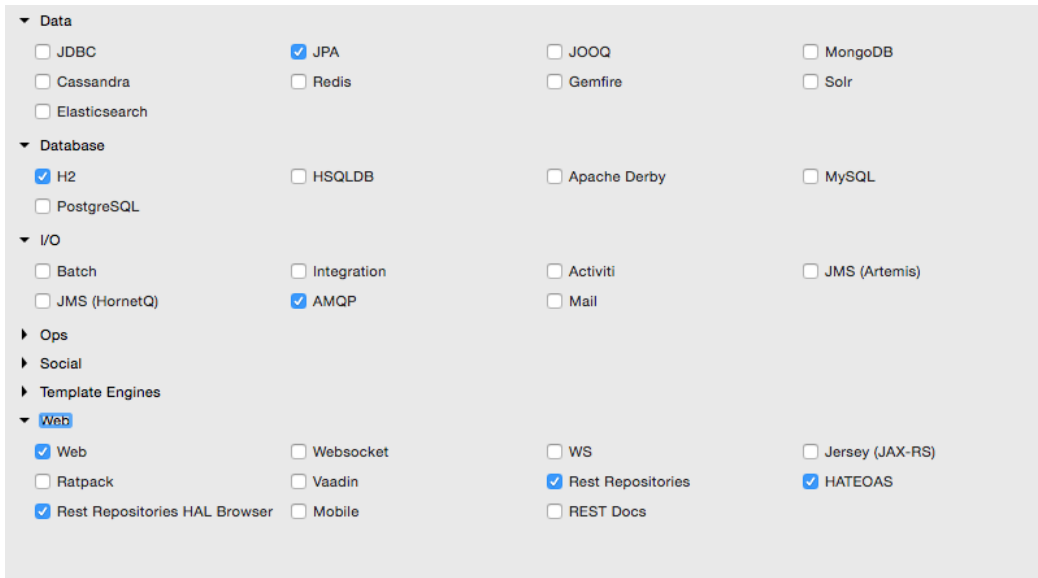
The Customer microservices class domain model diagram is as shown here:



`CustomerController` in the diagram is the REST endpoint, which invokes a component class, `CustomerComponent`. The component class/bean handles all the business logic. `CustomerRepository` is a Spring data JPA repository defined to handle the persistence of the `Customer` entity.

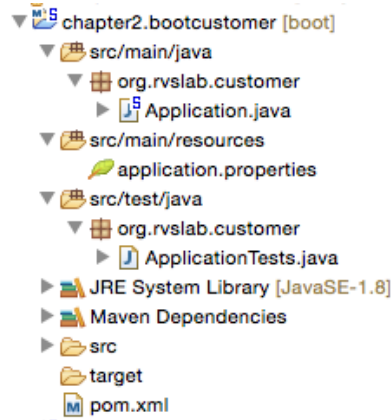
 The full source code of this example is available as the `chapter2.bootcustomer` and `chapter2.bootcustomernotification` projects in the code files of this book

1. Create a new Spring Boot project and call it `chapter2.bootcustomer`, the same way as earlier. Select the options as in the following screenshot in the starter module selection screen:



This will create a web project with JPA, the REST repository, and H2 as a database. H2 is a tiny in-memory embedded database with which it is easy to demonstrate database features. In the real world, it is recommended to use an appropriate enterprise-grade database. This example uses JPA to define persistence entities and the REST repository to expose REST-based repository services.

The project structure will be similar to the following screenshot:



2. Start building the application by adding an Entity class named `Customer`. For simplicity, there are only three fields added to the `Customer` Entity class: the autogenerated `id` field, `name`, and `email`. Take a look at the following code:

```
@Entity
class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private String email;
}
```

3. Add a repository class to handle the persistence handling of `Customer`. `CustomerRepository` extends the standard JPA repository. This means that all CRUD methods and default finder methods are automatically implemented by the Spring Data JPA repository, as follows:

```
@RepositoryRestResource
interface CustomerRepository extends JpaRepository<Customer, Long> {
    Optional<Customer> findByName(@Param("name") String name);
}
```

In this example, we added a new method to the repository class, `findByName`, which essentially searches the customer based on the customer name and returns a `Customer` object if there is a matching name.

4. The `@RepositoryRestResource` annotation enables the repository access through RESTful services. This will also enable HATEOAS and HAL by default. As for CRUD methods there is no additional business logic required, we will leave it as it is without controller or component classes. Using HATEOAS will help us navigate through Customer Repository methods effortlessly.

Note that there is no configuration added anywhere to point to any database. As H2 libraries are in the class path, all the configuration is done by default by Spring Boot based on the H2 autoconfiguration.

5. Update the `Application.java` file by adding `CommandLineRunner` to initialize the repository with some customer records, as follows:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    CommandLineRunner init(CustomerRepository repo) {
        return (evt) -> {
            repo.save(new Customer("Adam", "adam@boot.com"));
            repo.save(new Customer("John", "john@boot.com"));
            repo.save(new Customer("Smith", "smith@boot.com"));
            repo.save(new Customer("Edgar", "edgar@boot.com"));
            repo.save(new Customer("Martin", "martin@boot.com"));
            repo.save(new Customer("Tom", "tom@boot.com"));
            repo.save(new Customer("Sean", "sean@boot.com"));
        };
    }
}
```















6. `CommandLineRunner`, defined as a bean, indicates that it should run when it is contained in `SpringApplication`. This will insert six sample customer records into the database at startup.
7. At this point, run the application as Spring Boot App. Open the HAL browser and point the browser to `http://localhost:8080`.

8. In the **Explorer** section, point to `http://localhost:8080/customers` and click on **Go**. This will list all the customers in the **Response Body** section of the HAL browser.
9. In the **Explorer** section, enter `http://localhost:8080/customers?size=2&page=1&sort=name` and click on **Go**. This will automatically execute paging and sorting on the repository and return the result.

As the page size is set to 2 and the first page is requested, it will come back with two records in a sorted order.

10. Review the **Links** section. As shown in the following screenshot, it will facilitate navigating **first**, **next**, **prev**, and **last**. These are done using the HATEOAS links automatically generated by the repository browser:

Links

rel	title	name / index	docs	GET	NON-GET
first					
prev					
self					
next					
last					
profile					
search					

11. Also, one can explore the details of a customer by selecting the appropriate link, such as `http://localhost:8080/customers/2`.

12. As the next step, add a controller class, `CustomerController`, to handle service endpoints. There is only one endpoint in this class, `/register`, which is used to register a customer. If successful, it returns the `Customer` object as the response, as follows:

```
@RestController
class CustomerController{

    @Autowired
    CustomerRegistrar customerRegistrar;

    @RequestMapping( path="/register", method = RequestMethod.POST)
    Customer register(@RequestBody Customer customer){
        return customerRegistrar.register(customer);
    }
}
```

13. A `CustomerRegistrar` component is added to handle the business logic. In this case, there is only minimal business logic added to the component. In this component class, while registering a customer, we will just check whether the customer name already exists in the database or not. If it does not exist, then we will insert a new record, and otherwise, we will send an error message back, as follows:

```
@Component
class CustomerRegistrar {







    CustomerRespository customerRespository;

    @Autowired
    CustomerRegistrar(CustomerRespository customerRespository){
        this.customerRespository = customerRespository;
    }

    Customer register(Customer customer){
        Optional<Customer> existingCustomer = customerRespository.
        findByName(customer.getName());
        if (existingCustomer.isPresent()){
            throw new RuntimeException("is already exists");
        } else {
            customerRespository.save(customer);
        }
        return customer;
    }
}
```

- 14. Restart the Boot application and test using the HAL browser via the URL `http://localhost:8080`.
- 15. Point the **Explorer** field to `http://localhost:8080/customers`. Review the results in the **Links** section:

Links

rel	title	name / index	docs	GET	NON-GET
self					
profile					 Perform non-GET rec
search					

- 16. Click on the **NON-GET** option against **self**. This will open a form to create a new customer:

Create/Update ×

Customer

Name

Email

Action:

Make Request

- 17. Fill the form and change the **Action** as shown in the diagram. Click on the **Make Request** button. This will call the register service and register the customer. Try giving a duplicate name to test the negative case.

18. Let's complete the last part in the example by integrating the Customer Notification service to notify the customer. When registration is successful, send an e-mail to the customer by asynchronously calling the Customer Notification microservice
19. First update `CustomerRegistrar` to call the second service. This is done through messaging. In this case, we injected a `Sender` component to send a notification to the customer by passing the customer's e-mail address to the sender, as follows:

```

@Component
@Lazy
class CustomerRegistrar {

    CustomerRespository customerRespository;
    Sender sender;

    @Autowired
    CustomerRegistrar(CustomerRespository customerRespository,
Sender sender){
        this.customerRespository = customerRespository;
        this.sender = sender;
    }

    Customer register(Customer customer){
        Optional<Customer> existingCustomer = customerRespository.
findByName(customer.getName());
        if (existingCustomer.isPresent()){
            throw new RuntimeException("is already exists");
        } else {
            customerRespository.save(customer);
            sender.send(customer.getEmail());
        }
        return customer;
    }
}

```

20. The sender component will be based on RabbitMQ and AMQP. In this example, `RabbitMessagingTemplate` is used as explored in the last messaging example; take a look at the following:

```

@Component
@Lazy
class Sender {

    @Autowired

```

```
RabbitMessagingTemplate template;

@Bean
Queue queue() {
    return new Queue("CustomerQ", false);
}

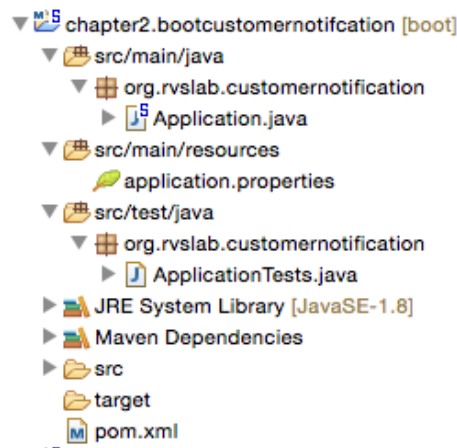
public void send(String message){
    template.convertAndSend("CustomerQ", message);
}
}
```

The `@Lazy` annotation is a useful one and it helps to increase the boot startup time. These beans will be initialized only when the need arises.

21. We will also update the `application.properties` file to include Rabbit MQ related properties, as follows:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

22. We are ready to send the message. To consume the message and send e-mails, we will create a notification service. For this, let's create another Spring Boot service, `chapter2.bootcustomernotification`. Make sure that the **AMQP** and **Mail** starter libraries are selected when creating the Spring Boot service. Both **AMQP** and **Mail** are under **I/O**.
23. The package structure of the `chapter2.bootcustomernotification` project is as shown here:



24. Add a Receiver class. The Receiver class waits for a message on customer. This will receive a message sent by the Customer Profile service. On the arrival of a message, it sends an e-mail, as follows:

```
@Component
class Receiver {
    @Autowired
    Mailer mailer;

    @Bean
    Queue queue() {
        return new Queue("CustomerQ", false);
    }

    @RabbitListener(queues = "CustomerQ")
    public void processMessage(String email) {
        System.out.println(email);
        mailer.sendMail(email);
    }
}
```

25. Add another component to send an e-mail to the customer. We will use JavaMailSender to send an e-mail via the following code:

```
@Component
class Mailer {
    @Autowired
    private JavaMailSender javaMailService;

    public void sendMail(String email){
        SimpleMailMessage mailMessage=new
            SimpleMailMessage();
        mailMessage.setTo(email);
        mailMessage.setSubject("Registration");
        mailMessage.setText("Successfully Registered");
        javaMailService.send(mailMessage);
    }
}
```

Behind the scenes, Spring Boot automatically configures all the parameters required by JavaMailSender.

26. To test SMTP, a test setup for SMTP is required to ensure that the mails are going out. In this example, FakeSMTP will be used. You can download FakeSMTP from <http://nilhcem.github.io/FakeSMTP>.

27. Once you download `fakeSMTP-2.0.jar`, run the SMTP server by executing the following command:

```
$ java -jar fakeSMTP-2.0.jar
```

This will open a GUI to monitor e-mail messages. Click on the **Start Server** button next to the listening port textbox.

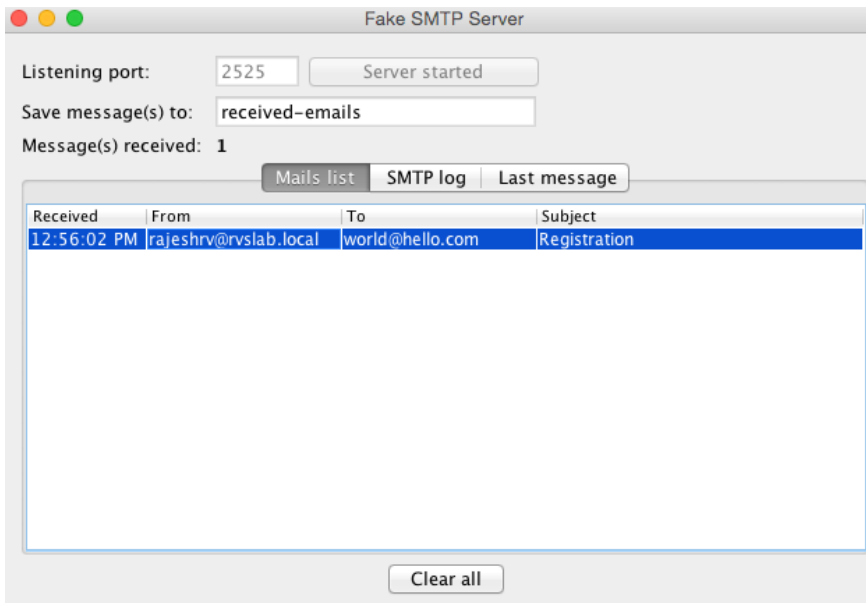
28. Update `application.properties` with the following configuration parameters to connect to RabbitMQ as well as to the mail server:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

```
spring.mail.host=localhost
spring.mail.port=2525
```

29. We are ready to test our microservices end to end. Start both the Spring Boot apps. Open the browser and repeat the customer creation steps through the HAL browser. In this case, immediately after submitting the request, we will be able to see the e-mail in the SMTP GUI.


Internally, the Customer Profile service asynchronously calls the Customer Notification service, which, in turn, sends the e-mail message to the SMTP server:



Spring Boot actuators

The previous sections explored most of the Spring Boot features required to develop a microservice. In this section, some of the production-ready operational aspects of Spring Boot will be explored.























Spring Boot actuators provide an excellent out-of-the-box mechanism to monitor and manage Spring Boot applications in production:

 The full source code of this example is available as the `chapter2.bootactuator` project in the code files of this book

1. Create another **Spring Starter Project** and name it `chapter2.bootactuator`. This time, select **Web** and **Actuators** under **Ops**. Similar to the `chapter2.bootrest` project, add a `GreeterController` endpoint with the `greet` method.
2. Start the application as Spring Boot app.
3. Point the browser to `localhost:8080/actuator`. This will open the HAL browser. Then, review the **Links** section.

A number of links are available under the **Links** section. These are automatically exposed by the Spring Boot actuator:

Links

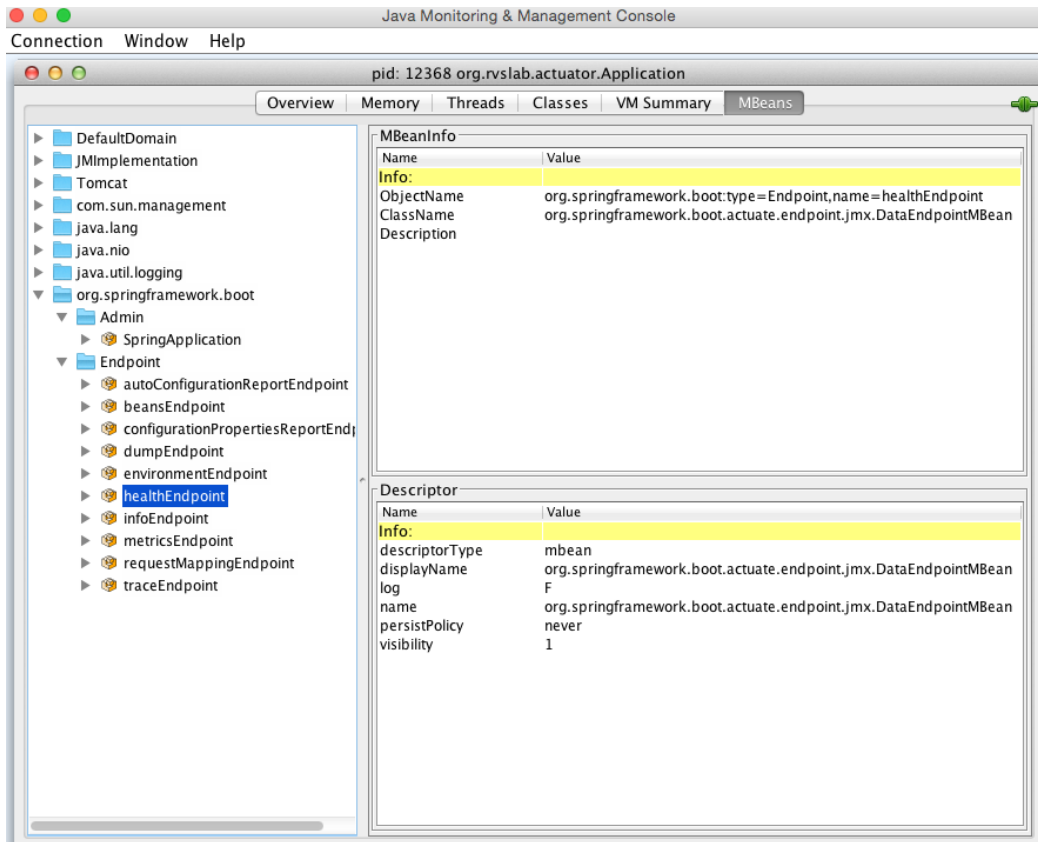
rel	title	name / index	docs	GET	NON-GET
self					
dump					
configprops					
env					
mappings					
info					
health					
autoconfig					
metrics					
trace					
beans					

Some of the important links are listed as follows:

- `dump`: This performs a thread dump and displays the result
- `mappings`: This lists all the HTTP request mappings
- `info`: This displays information about the application
- `health`: This displays the application's health conditions
- `autoconfig`: This displays the autoconfiguration report
- `metrics`: This shows different metrics collected from the application

Monitoring using JConsole

Alternately, we can use the JMX console to see the Spring Boot information. Connect to the remote Spring Boot instance from JConsole. The Boot information will be shown as follows:



Monitoring using SSH

Spring Boot provides remote access to the Boot application using SSH. The following command connects to the Spring Boot application from a terminal window:

```
$ ssh -p 2000 user@localhost
```

The password can be customized by adding the `shell.auth.simple.user.password` property in the `application.properties` file. The updated `application.properties` file will look similar to the following

```
shell.auth.simple.user.password=admin
```

When connected with the preceding command, similar actuator information can be accessed. Here is an example of the metrics information accessed through the CLI:

- `help`: This lists out all the options available
- `dashboard`: This is one interesting feature that shows a lot of system-level information

Configuring application information

The following properties can be set in `application.properties` to customize application-related information. After adding, restart the server and visit the `/info` endpoint of the actuator to take a look at the updated information, as follows:

```
info.app.name=Boot actuator
info.app.description= My Greetings Service
info.app.version=1.0.0
```

Adding a custom health module

Adding a new custom module to the Spring Boot application is not so complex. To demonstrate this feature, assume that if a service gets more than two transactions in a minute, then the server status will be set as Out of Service.

In order to customize this, we have to implement the `HealthIndicator` interface and override the `health` method. The following is a quick and dirty implementation to do the job:

```
class TPSCounter {
    LongAdder count;
    int threshold = 2;
```

```
Calendar expiry = null;

TPSCounter(){
    this.count = new LongAdder();
    this.expiry = Calendar.getInstance();
    this.expiry.add(Calendar.MINUTE, 1);
}

boolean isExpired(){
    return Calendar.getInstance().after(expiry);
}

boolean isWeak(){
    return (count.intValue() > threshold);
}

void increment(){
    count.increment();
}
}
```

The preceding class is a simple POJO class that maintains the transaction counts in the window. The `isWeak` method checks whether the transaction in a particular window reached its threshold. The `isExpired` method checks whether the current window is expired or not. The `increment` method simply increases the counter value.

For the next step, implement our custom health indicator class, `TPSHealth`. This is done by extending `HealthIndicator`, as follows:

```
@Component
class TPSHealth implements HealthIndicator {
    TPSCounter counter;

    @Override
    public Health health() {
        boolean health = counter.isWeak(); // perform some specific
health check
        if (health) {
            return Health.outOfService().withDetail("Too many
requests", "OutOfService").build();
        }
        return Health.up().build();
    }
}
```

```

void updateTx() {
    if(counter == null || counter.isExpired()){
        counter = new TPSCounter();
    }
    counter.increment();
}
}

```

The `health` method checks whether the counter is weak or not. A weak counter means the service is handling more transactions than it can handle. If it is weak, it marks the instance as Out of Service.

Finally, we will autowire `TPSHealth` into the `GreetingController` class and then call `health.updateTx()` in the `greet` method, as follows:

```

Greet greet() {
    logger.info("Serving Request....!!!");
    health.updateTx();
    return new Greet("Hello World!");
}

```

Go to the `/health` end point in the HAL browser and take a look at the status of the server.

Now, open another browser, point to `http://localhost:8080`, and fire the service twice or thrice. Go back to the `/health` endpoint and refresh to see the status. It should be changed to Out of Service.

In this example, as there is no action taken other than collecting the health status, even though the status is Out of Service, new service calls will still go through. However, in the real world, a program should read the `/health` endpoint and block further requests from going to this instance.

Building custom metrics

Similar to `health`, customization of the metrics is also possible. The following example shows how to add counter service and gauge service, just for demonstration purposes:

```

@Autowired
CounterService counterService;

@Autowired
GaugeService gaugeService;

```

Add the following methods in the greet method:

```
this.counterService.increment("greet.txnCount");  
this.gaugeService.submit("greet.customgauge", 1.0);
```

Restart the server and go to `/metrics` to see the new gauge and counter added already reflected there.

Documenting microservices

The traditional approach of API documentation is either by writing service specification documents or using static service registries. With a large number of microservices, it would be hard to keep the documentation of APIs in sync.

Microservices can be documented in many ways. This section will explore how microservices can be documented using the popular Swagger framework. The following example will use Springfox libraries to generate REST API documentation. Springfox is a set of Java- and Spring-friendly libraries.

Create a new **Spring Starter Project** and select **Web** in the library selection window. Name the project `chapter2.swagger`.



The full source code of this example is available as the `chapter2.swagger` project in the code files of this book

As Springfox libraries are not part of the Spring suite, edit `pom.xml` and add Springfox Swagger library dependencies. Add the following dependencies to the project:

```
<dependency>  
  <groupId>io.springfox</groupId>  
  <artifactId>springfox-swagger2</artifactId>  
  <version>2.3.1</version>  
</dependency>  
<dependency>  
  <groupId>io.springfox</groupId>  
  <artifactId>springfox-swagger-ui</artifactId>  
  <version>2.3.1</version>  
</dependency>
```

Create a REST service similar to the services created earlier, but also add the `@EnableSwagger2` annotation, as follows:

```
@SpringBootApplication
@EnableSwagger2
public class Application {
```

This is all that's required for a basic Swagger documentation. Start the application and point the browser to `http://localhost:8080/swagger-ui.html`. This will open the Swagger API documentation page:

Api Documentation
Api Documentation

Created by Contact Email
[Apache 2.0](#)

basic-error-controller : Basic Error Controller [Show/Hide](#) [List Operations](#) [Expand Operations](#)

greet-controller : Greet Controller [Show/Hide](#) [List Operations](#) [Expand Operations](#)

DELETE	/	greet
GET	/	greet
HEAD	/	greet
OPTIONS	/	greet
PATCH	/	greet
POST	/	greet
PUT	/	greet

[BASE URL: / , API VERSION: 1.0]

As shown in the diagram, the Swagger lists out the possible operations on **Greet Controller**. Click on the **GET** operation. This expands the **GET** row, which provides an option to try out the operation.

Summary

In this chapter, you learned about Spring Boot and its key features to build production-ready applications.

We explored the previous-generation web applications and then how Spring Boot makes developers' lives easier to develop fully qualified microservices. We also discussed the asynchronous message-based interaction between services. Further, we explored how to achieve some of the key capabilities required for microservices, such as security, HATEOAS, cross-origin, configurations, and so on with practical examples. We also took a look at how Spring Boot actuators help the operations teams and also how we can customize it to our needs. Finally, documenting microservices APIs was also explored.

In the next chapter, we will take a deeper look at some of the practical issues that may arise when implementing microservices. We will also discuss a capability model that essentially helps organizations when dealing with large microservices implementations.

3

Applying Microservices Concepts

Microservices are good, but can also be an evil if they are not properly conceived. Wrong microservice interpretations could lead to irrecoverable failures.

This chapter will examine the technical challenges around practical implementations of microservices. It will also provide guidelines around critical design decisions for successful microservices development. The solutions and patterns for a number of commonly raised concerns around microservices will also be examined. This chapter will also review the challenges in enterprise scale microservices development, and how to overcome those challenges. More importantly, a capability model for a microservices ecosystem will be established at the end.

In this chapter you will learn about the following:

- Trade-offs between different design choices and patterns to be considered when developing microservices
- Challenges and anti-patterns in developing enterprise grade microservices
- A capability model for a microservices ecosystem

Patterns and common design decisions

Microservices have gained enormous popularity in recent years. They have evolved as the preferred choice of architects, putting SOA into the backyards. While acknowledging the fact that microservices are a vehicle for developing scalable cloud native systems, successful microservices need to be carefully designed to avoid catastrophes. Microservices are not the one-size-fits-all, universal solution for all architecture problems.


Generally speaking, microservices are a great choice for building a lightweight, modular, scalable, and distributed system of systems. Over-engineering, wrong use cases, and misinterpretations could easily turn the system into a disaster. While selecting the right use cases is paramount in developing a successful microservice, it is equally important to take the right design decisions by carrying out an appropriate trade-off analysis. A number of factors are to be considered when designing microservices, as detailed in the following sections.

Establishing appropriate microservice boundaries

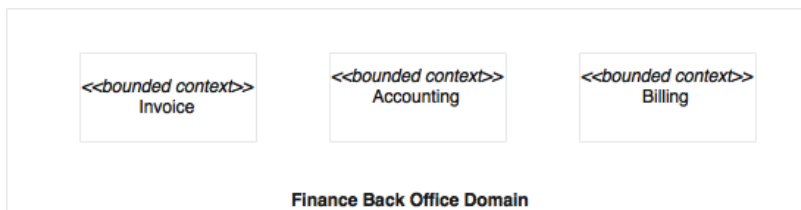
One of the most common questions relating to microservices is regarding the size of the service. How big (mini-monolithic) or how small (nano service) can a microservice be, or is there anything like right-sized services? Does size really matter?

A quick answer could be "one REST endpoint per microservice", or "less than 300 lines of code", or "a component that performs a single responsibility". But before we pick up any of these answers, there is lot more analysis to be done to understand the boundaries for our services.

Domain-driven design (DDD) define the concept of a **bounded context**. A bounded context is a subdomain or a subsystem of a larger domain or system that is responsible for performing a particular function.

[ Read more about DDD at <http://domainlanguage.com/ddd/>.]

The following diagram is an example of the domain model:



In a finance back office, system invoices, accounting, billing, and the like represent different bounded contexts. These bounded contexts are strongly isolated domains that are closely aligned with business capabilities. In the financial domain, the invoices, accounting, and billing are different business capabilities often handled by different subunits under the finance department.

A bounded context is a good way to determine the boundaries of microservices. Each bounded context could be mapped to a single microservice. In the real world, communication between bounded contexts are typically less coupled, and often, disconnected.

Even though real world organizational boundaries are the simplest mechanisms for establishing a bounded context, these may prove wrong in some cases due to inherent problems within the organization's structures. For example, a business capability may be delivered through different channels such as front offices, online, roaming agents, and so on. In many organizations, the business units may be organized based on delivery channels rather than the actual underlying business capabilities. In such cases, organization boundaries may not provide accurate service boundaries.

A top-down domain decomposition could be another way to establish the right bounded contexts.

There is no silver bullet to establish microservices boundaries, and often, this is quite challenging. Establishing boundaries is much easier in the scenario of monolithic application to microservices migration, as the service boundaries and dependencies are known from the existing system. On the other hand, in a green field microservices development, the dependencies are hard to establish upfront

The most pragmatic way to design microservices boundaries is to run the scenario at hand through a number of possible options, just like a service litmus test. Keep in mind that there may be multiple conditions matching a given scenario that will lead to a trade-off analysis.

The following scenarios could help in defining the microservice boundaries

Autonomous functions

If the function under review is autonomous by nature, then it can be taken as a microservices boundary. Autonomous services typically would have fewer dependencies on external functions. They accept input, use its internal logic and data for computation, and return a result. All utility functions such as an encryption engine or a notification engine are straightforward candidates

A delivery service that accepts an order, processes it, and then informs the trucking service is another example of an autonomous service. An online flight search based on cached seat availability information is yet another example of an autonomous function.

Size of a deployable unit

Most of the microservices ecosystems will take advantage of automation, such as automatic integration, delivery, deployment, and scaling. Microservices covering broader functions result in larger deployment units. Large deployment units pose challenges in automatic file copy, file download, deployment, and start up times. For instance, the size of a service increases with the density of the functions that it implements.

A good microservice ensures that the size of its deployable units remains manageable.

Most appropriate function or subdomain

It is important to analyze what would be the most useful component to detach from the monolithic application. This is particularly applicable when breaking monolithic applications into microservices. This could be based on parameters such as resource-intensiveness, cost of ownership, business benefits, or flexibility.

In a typical hotel booking system, approximately 50-60% of the requests are search-based. In this case, moving out the search function could immediately bring in flexibility, business benefits, cost reduction, resource free up, and so on.

Polyglot architecture

One of the key characteristics of microservices is its support for polyglot architecture. In order to meet different non-functional and functional requirements, components may require different treatments. It could be different architectures, different technologies, different deployment topologies, and so on. When components are identified, review them against the requirement for polyglot architectures.

In the hotel booking scenario mentioned earlier, a Booking microservice may need transactional integrity, whereas a Search microservice may not. In this case, the Booking microservice may use an ACID compliance database such as MySQL, whereas the Search microservice may use an eventual consistent database such as Cassandra.

Selective scaling

Selective scaling is related to the previously discussed polyglot architecture. In this context, all functional modules may not require the same level of scalability. Sometimes, it may be appropriate to determine boundaries based on scalability requirements.

For example, in the hotel booking scenario, the Search microservice has to scale considerably more than many of the other services such as the Booking microservice or the Notification microservice due to the higher velocity of search requests. In this case, a separate Search microservice could run on top of an Elasticsearch or an in-memory data grid for better response.

Small, agile teams

Microservices enable Agile development with small, focused teams developing different parts of the pie. There could be scenarios where parts of the systems are built by different organizations, or even across different geographies, or by teams with varying skill sets. This approach is a common practice, for example, in manufacturing industries.

In the microservices world, each of these teams builds different microservices, and then assembles them together. Though this is not the desired way to break down the system, organizations may end up in such situations. Hence, this approach cannot be completely ruled out.

In an online product search scenario, a service could provide personalized options based on what the customer is looking for. This may require complex machine learning algorithms, and hence need a specialist team. In this scenario, this function could be built as a microservice by a separate specialist team.

Single responsibility

In theory, the single responsibility principle could be applied at a method, at a class, or at a service. However, in the context of microservices, it does not necessarily map to a single service or endpoint.

A more practical approach could be to translate single responsibility into single business capability or a single technical capability. As per the *single responsibility* principle, one responsibility cannot be shared by multiple microservices. Similarly, one microservice should not perform multiple responsibilities.

There could, however, be special cases where a single business capability is divided across multiple services. One of such cases is managing the customer profile, where there could be situations where you may use two different microservices for managing reads and writes using a **Command Query Responsibility Segregation (CQRS)** approach to achieve some of the quality attributes.

Replicability or changeability

Innovation and speed are of the utmost importance in IT delivery. Microservices boundaries should be identified in such a way that each microservice is easily detachable from the overall system, with minimal cost of re-writing. If part of the system is just an experiment, it should ideally be isolated as a microservice.

An organization may develop a recommendation engine or a customer ranking engine as an experiment. If the business value is not realized, then throw away that service, or replace it with another one.

Many organizations follow the startup model, where importance is given to meeting functions and quick delivery. These organizations may not worry too much about the architecture and technologies. Instead, the focus will be on what tools or technologies can deliver solutions faster. Organizations increasingly choose the approach of developing **Minimum Viable Products (MVPs)** by putting together a few services, and allowing the system to evolve. Microservices play a vital role in such cases where the system evolves, and services gradually get rewritten or replaced.

Coupling and cohesion

Coupling and cohesion are two of the most important parameters for deciding service boundaries. Dependencies between microservices have to be evaluated carefully to avoid highly coupled interfaces. A functional decomposition, together with a modeled dependency tree, could help in establishing a microservices boundary. Avoiding too chatty services, too many synchronous request-response calls, and cyclic synchronous dependencies are three key points, as these could easily break the system. A successful equation is to keep high cohesion within a microservice, and loose coupling between microservices. In addition to this, ensure that transaction boundaries are not stretched across microservices. A first class microservice will react upon receiving an event as an input, execute a number of internal functions, and finally send out another event. As part of the compute function, it may read and write data to its own local store.

Think microservice as a product

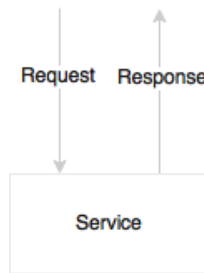
DDD also recommends mapping a bounded context to a product. As per DDD, each bounded context is an ideal candidate for a product. Think about a microservice as a product by itself. When microservice boundaries are established, assess them from a product's point of view to see whether they really stack up as product. It is much easier for business users to think boundaries from a product point of view. A product boundary may have many parameters, such as a targeted community, flexibility in deployment, sell-ability, reusability, and so on

Designing communication styles

Communication between microservices can be designed either in synchronous (request-response) or asynchronous (fire and forget) styles

Synchronous style communication

The following diagram shows an example request/response style service:



In synchronous communication, there is no shared state or object. When a caller requests a service, it passes the required information and waits for a response. This approach has a number of advantages.

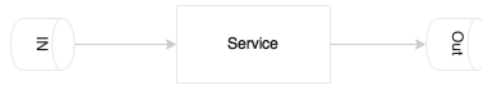
An application is stateless, and from a high availability standpoint, many active instances can be up and running, accepting traffic. Since there are no other infrastructure dependencies such as a shared messaging server, there are management fewer overheads. In case of an error at any stage, the error will be propagated back to the caller immediately, leaving the system in a consistent state, without compromising data integrity.

The downside in a synchronous request-response communication is that the user or the caller has to wait until the requested process gets completed. As a result, the calling thread will wait for a response, and hence, this style could limit the scalability of the system.

A synchronous style adds hard dependencies between microservices. If one service in the service chain fails, then the entire service chain will fail. In order for a service to succeed, all dependent services have to be up and running. Many of the failure scenarios have to be handled using timeouts and loops.

Asynchronous style communication

The following diagram is a service designed to accept an asynchronous message as input, and send the response asynchronously for others to consume:



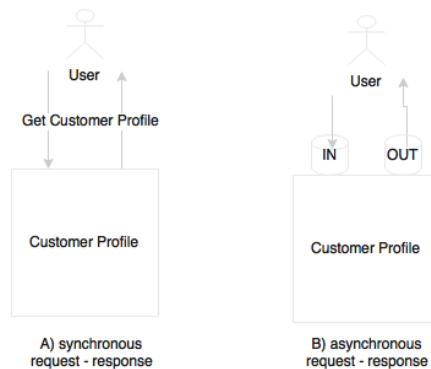
The asynchronous style is based on reactive event loop semantics which decouple microservices. This approach provides higher levels of scalability, because services are independent, and can internally spawn threads to handle an increase in load. When overloaded, messages will be queued in a messaging server for later processing. That means that if there is a slowdown in one of the services, it will not impact the entire chain. This provides higher levels of decoupling between services, and therefore maintenance and testing will be simpler.

The downside is that it has a dependency to an external messaging server. It is complex to handle the fault tolerance of a messaging server. Messaging typically works with an active/passive semantics. Hence, handling continuous availability of messaging systems is harder to achieve. Since messaging typically uses persistence, a higher level of I/O handling and tuning is required.

How to decide which style to choose?

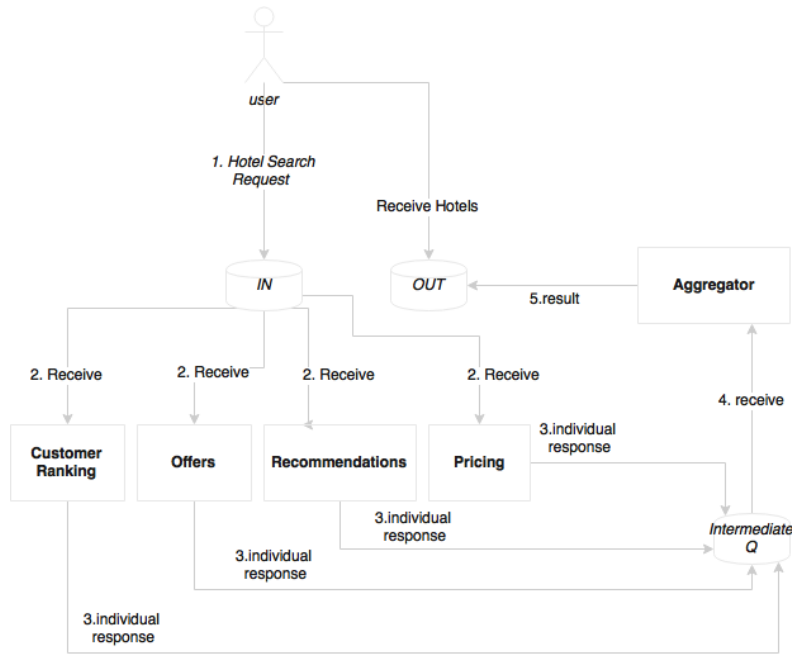
Both approaches have their own merits and constraints. It is not possible to develop a system with just one approach. A combination of both approaches is required based on the use cases. In principle, the asynchronous approach is great for building true, scalable microservice systems. However, attempting to model everything as asynchronous leads to complex system designs.

How does the following example look in the context where an end user clicks on a UI to get profile details



This is perhaps a simple query to the backend system to get a result in a request-response model. This can also be modeled in an asynchronous style by pushing a message to an input queue, and waiting for a response in an output queue till a response is received for the given correlation ID. However, though we use asynchronous messaging, the user is still blocked for the entire duration of the query.

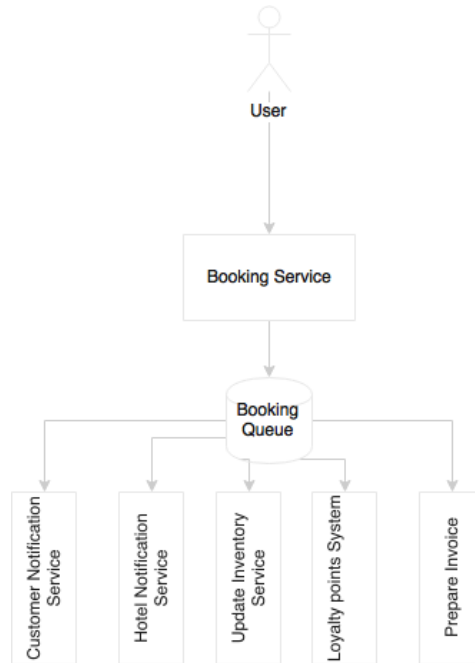
Another use case is that of a user clicking on a UI to search hotels, which is depicted in the following diagram:



This is very similar to the previous scenario. However, in this case, we assume that this business function triggers a number of activities internally before returning the list of hotels back to the user. For example, when the system receives this request, it calculates the customer ranking, gets offers based on the destination, gets recommendations based on customer preferences, optimizes the prices based on customer values and revenue factors, and so on. In this case, we have an opportunity to do many of these activities in parallel so that we can aggregate all these results before presenting them to the customer. As shown in the preceding diagram, virtually any computational logic could be plugged in to the search pipeline listening to the **IN** queue.

An effective approach in this case is to start with a synchronous request response, and refactor later to introduce an asynchronous style when there is value in doing that.

The following example shows a fully asynchronous style of service interactions:



The service is triggered when the user clicks on the booking function. It is again, by nature, a synchronous style communication. When booking is successful, it sends a message to the customer's e-mail address, sends a message to the hotel's booking system, updates the cached inventory, updates the loyalty points system, prepares an invoice, and perhaps more. Instead of pushing the user into a long wait state, a better approach is to break the service into pieces. Let the user wait till a booking record is created by the Booking Service. On successful completion, a booking event will be published, and return a confirmation message back to the user. Subsequently, all other activities will happen in parallel, asynchronously.

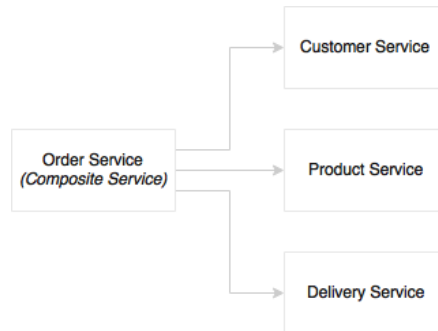
In all three examples, the user has to wait for a response. With the new web application frameworks, it is possible to send requests asynchronously, and define the callback method, or set an observer for getting a response. Therefore, the users won't be fully blocked from executing other activities.

In general, an asynchronous style is always better in the microservices world, but identifying the right pattern should be purely based on merits. If there are no merits in modeling a transaction in an asynchronous style, then use the synchronous style till you find an appealing case. Use reactive programming frameworks to avoid complexity when modeling user-driven requests, modeled in an asynchronous style.

Orchestration of microservices

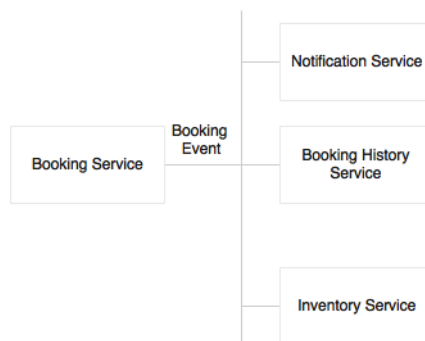
Composability is one of the service design principles. This leads to confusion around who is responsible for the composing services. In the SOA world, ESBs are responsible for composing a set of finely-grained services. In some organizations, ESBs play the role of a proxy, and service providers themselves compose and expose coarse-grained services. In the SOA world, there are two approaches for handling such situations.

The first approach is orchestration, which is depicted in the following diagram



In the orchestration approach, multiple services are stitched together to get a complete function. A central brain acts as the orchestrator. As shown in the diagram, the order service is a composite service that will orchestrate other services. There could be sequential as well as parallel branches from the master process. Each task will be fulfilled by an atomic task service, typically a web service. In the SOA world, ESBs play the role of orchestration. The orchestrated service will be exposed by ESBs as a composite service.

The second approach is choreography, which is shown in the following diagram:

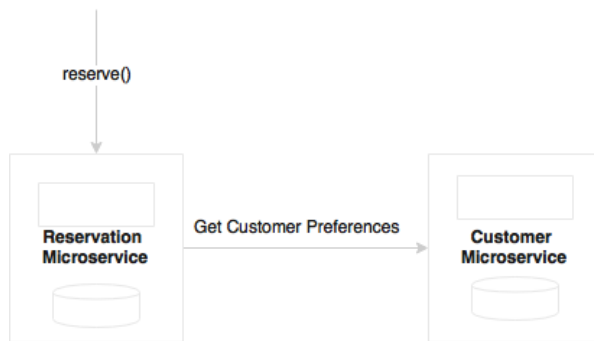


In the choreography approach, there is no central brain. An event, a booking event in this case, is published by a producer, a number of consumers wait for the event, and independently apply different logics on the incoming event. Sometimes, events could even be nested where the consumers can send another event which will be consumed by another service. In the SOA world, the caller pushes a message to the ESB, and the downstream flow will be automatically determined by the consuming services

Microservices are autonomous. This essentially means that in an ideal situation, all required components to complete their function should be within the service. This includes the database, orchestration of its internal services, intrinsic state management, and so on. The service endpoints provide coarse-grained APIs. This is perfectly fine as long as there are no external touch points required. But in reality, microservices may need to talk to other microservices to fulfil their function

In such cases, choreography is the preferred approach for connecting multiple microservices together. Following the autonomy principle, a component sitting outside a microservice and controlling the flow is not the desired option. If the use case can be modeled in choreographic style, that would be the best possible way to handle the situation.

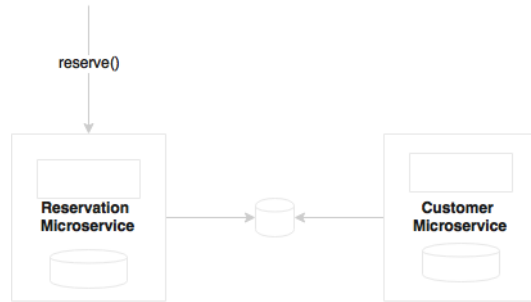
But it may not be possible to model choreography in all cases. This is depicted in the following diagram:



In the preceding example, Reservation and Customer are two microservices, with clearly segregated functional responsibilities. A case could arise when Reservation would want to get Customer preferences while creating a reservation. These are quite normal scenarios when developing complex systems.

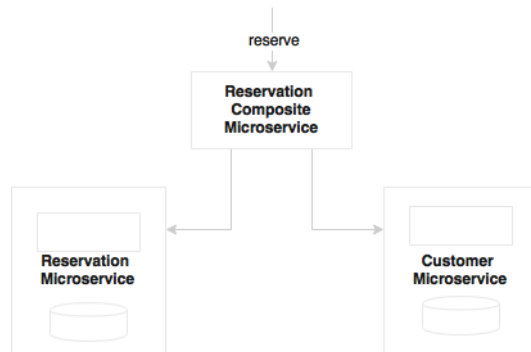
Can we move Customer to Reservation so that Reservation will be complete by itself? If Customer and Reservation are identified as two microservices based on various factors, it may not be a good idea to move Customer to Reservation. In such a case, we will meet another monolithic application sooner or later.

Can we make the Reservation to Customer call asynchronous? This example is shown in the following diagram:



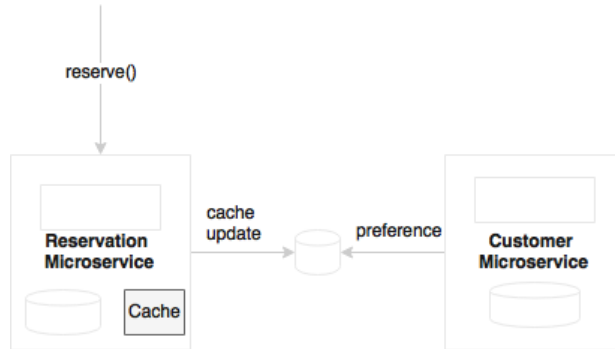
Customer preference is required for Reservation to progress, and hence, it may require a synchronous blocking call to Customer. Retrofitting this by modeling asynchronously does not really make sense.

Can we take out just the orchestration bit, and create another composite microservice, which then composes Reservation and Customer?



This is acceptable in the approach for composing multiple components within a microservice. But creating a composite microservice may not be a good idea. We will end up creating many microservices with no business alignment, which would not be autonomous, and could result in many fine-grained microservices

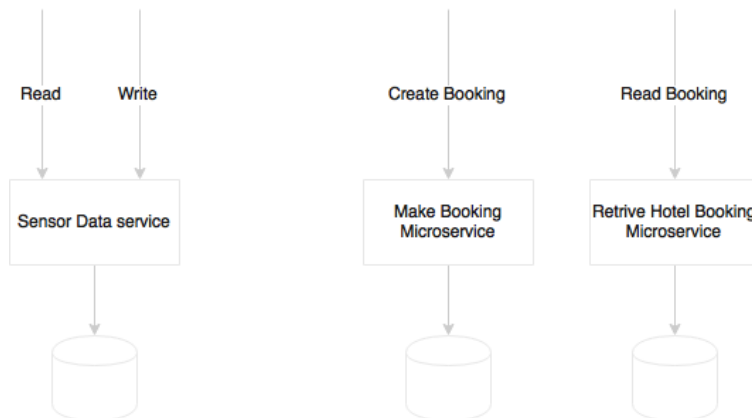
Can we duplicate customer preference by keeping a slave copy of the preference data into Reservation?



Changes will be propagated whenever there is a change in the master. In this case, Reservation can use customer preference without fanning out a call. It is a valid thought, but we need to carefully analyze this. Today we replicate customer preference, but in another scenario, we may want to reach out to customer service to see whether the customer is black-listed from reserving. We have to be extremely careful in deciding what data to duplicate. This could add to the complexity.

How many endpoints in a microservice?

In many situations, developers are confused with the number of endpoints per microservice. The question really is whether to limit each microservice with one endpoint or multiple endpoints:



The number of endpoints is not really a decision point. In some cases, there may be only one endpoint, whereas in some other cases, there could be more than one endpoint in a microservice. For instance, consider a sensor data service which collects sensor information, and has two logical endpoints: create and read. But in order to handle CQRS, we may create two separate physical microservices as shown in the case of **Booking** in the preceding diagram. Polyglot architecture could be another scenario where we may split endpoints into different microservices.

Considering a notification engine, notifications will be sent out in response to an event. The process of notification such as preparation of data, identification of person, and delivery mechanisms, are different for different events. Moreover, we may want to scale each of these processes differently at different time windows. In such situations, we may decide to break each notification endpoint into a separate microservice.

In yet another example, a Loyalty Points microservice may have multiple services such as accrue, redeem, transfer, and balance. We may not want to treat each of these services differently. All of these services are connected and use the points table for data. If we go with one endpoint per service, we will end up in a situation where many fine-grained services access data from the same data store or replicated copies of the same data store.

In short, the number of endpoints is not a design decision. One microservice may host one or more endpoints. Designing appropriate bounded context for a microservice is more important.

One microservice per VM or multiple?

One microservice could be deployed in multiple **Virtual Machines (VMs)** by replicating the deployment for scalability and availability. This is a no brainer. The question is whether multiple microservices could be deployed in one virtual machine? There are pros and cons for this approach. This question typically arises when the services are simple, and the traffic volume is less.

Consider an example where we have a couple of microservices, and the overall transaction per minute is less than 10. Also assume that the smallest possible VM size available is 2-core 8 GB RAM. A further assumption is that in such cases, a 2-core 8 GB VM can handle 10-15 transactions per minute without any performance concerns. If we use different VMs for each microservice, it may not be cost effective, and we will end up paying more for infrastructure and license, since many vendors charge based on the number of cores.

The simplest way to approach this problem is to ask a few questions:

- Does the VM have enough capacity to run both services under peak usage?
- Do we want to treat these services differently to achieve SLAs (selective scaling)? For example, for scalability, if we have an all-in-one VM, we will have to replicate VMs which replicate all services.
- Are there any conflicting resource requirements? For example, different OS versions, JDK versions, and others.

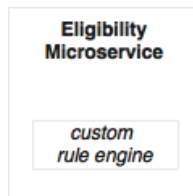
If all your answers are *No*, then perhaps we can start with collocated deployment, until we encounter a scenario to change the deployment topology. However, we will have to ensure that these services are not sharing anything, and are running as independent OS processes.

Having said that, in an organization with matured virtualized infrastructure or cloud infrastructure, this may not be a huge concern. In such environments, the developers need not worry about where the services are running. Developers may not even think about capacity planning. Services will be deployed in a compute cloud. Based on the infrastructure availability, SLAs and the nature of the service, the infrastructure self-manages deployments. AWS Lambda is a good example of such a service.

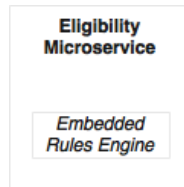
Rules engine – shared or embedded?

Rules are an essential part of any system. For example, an offer eligibility service may execute a number of rules before making a yes or no decision. Either we hand code rules, or we may use a rules engine. Many enterprises manage rules centrally in a rules repository as well as execute them centrally. These enterprise rule engines are primarily used for providing the business an opportunity to author and manage rules as well as reuse rules from the central repository. **Drools** is one of the popular open source rules engines. IBM, FICO, and Bosch are some of the pioneers in the commercial space. These rule engines improve productivity, enable reuse of rules, facts, vocabularies, and provide faster rule execution using the rete algorithm.

In the context of microservices, a central rules engine means fanning out calls from microservices to the central rules engine. This also means that the service logic is now in two places, some within the service, and some external to the service. Nevertheless, the objective in the context of microservices is to reduce external dependencies:



If the rules are simple enough, few in numbers, only used within the boundaries of a service, and not exposed to business users for authoring, then it may be better to hand-code business rules than rely on an enterprise rule engine:



If the rules are complex, limited to a service context, and not given to business users, then it is better to use an embedded rules engine within the service:



If the rules are managed and authored by business, or if the rules are complex, or if we are reusing rules from other service domains, then a central authoring repository with a locally embedded execution engine could be a better choice.

Note that this has to be carefully evaluated since all vendors may not support the local rule execution approach, and there could be technology dependencies such as running rules only within a specific application server, and so on

Role of BPM and workflows

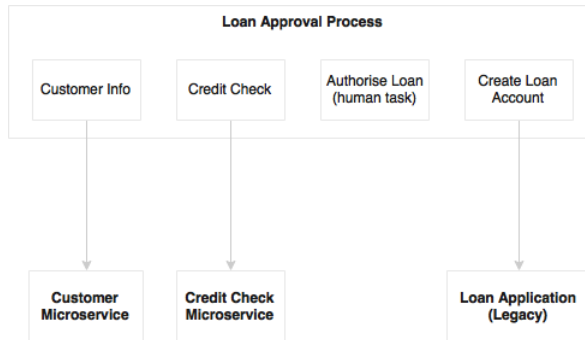
Business Process Management (BPM) and **Intelligent Business Process Management (iBPM)** are tool suites for designing, executing, and monitoring business processes.

Typical use cases for BPM are:

- Coordinating a long-running business process, where some processes are realized by existing assets, whereas some other areas may be niche, and there is no concrete implementation of the processes being in place. BPM allows composing both types, and provides an end-to-end automated process. This often involves systems and human interactions.

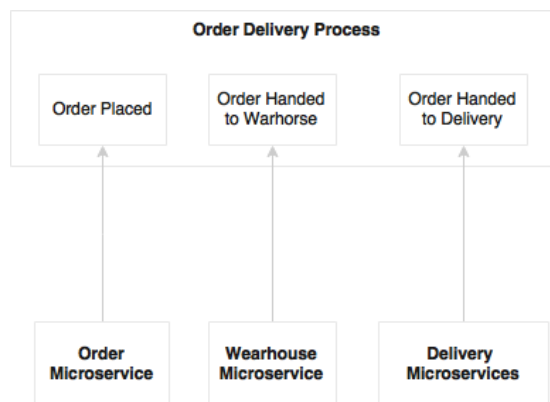
- Process-centric organizations, such as those that have implemented Six Sigma, want to monitor their processes for continuous improvement on efficiency.
- Process re-engineering with a top-down approach by redefining the business process of an organization.

There could be two scenarios where BPM fits in the microservices world



The first scenario is business process re-engineering, or threading an end-to-end long running business process, as stated earlier. In this case, BPM operates at a higher level, where it may automate a cross-functional, long-running business process by stitching a number of coarse-grained microservices, existing legacy connectors, and human interactions. As shown in the preceding diagram, the loan approval BPM invokes microservices as well as legacy application services. It also integrates human tasks.

In this case, microservices are headless services that implement a subprocess. From the microservices' perspective, BPM is just another consumer. Care needs to be taken in this approach to avoid accepting a shared state from a BPM process as well as moving business logic to BPM:

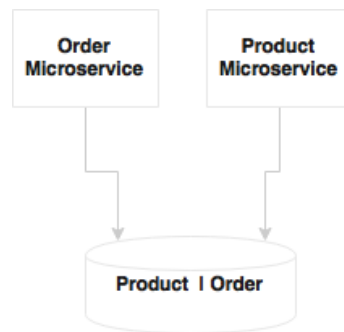


The second scenario is monitoring processes, and optimizing them for efficiency. This goes hand in hand with a completely automated, asynchronously choreographed microservices ecosystem. In this case, microservices and BPM work as independent ecosystems. Microservices send events at various timeframes such as the start of a process, state changes, end of a process, and so on. These events are used by the BPM engine to plot and monitor process states. We may not require a full-fledged BPM solution for this, as we are only mocking a business process to monitor its efficiency. In this case, the order delivery process is not a BPM implementation, but it is more of a monitoring dashboard that captures and displays the progress of the process.

To summarize, BPM could still be used at a higher level for composing multiple microservices in situations where end-to-end cross-functional business processes are modeled by automating systems and human interactions. A better and simpler approach is to have a business process dashboard to which microservices feed state change events as mentioned in the second scenario.

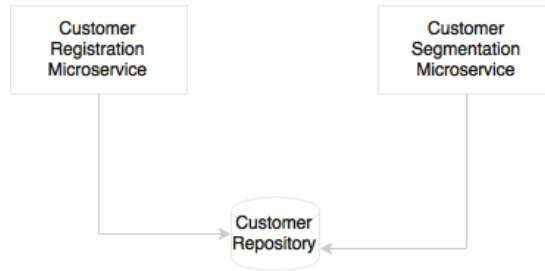
Can microservices share data stores?

In principle, microservices should abstract presentation, business logic, and data stores. If the services are broken as per the guidelines, each microservice logically could use an independent database:

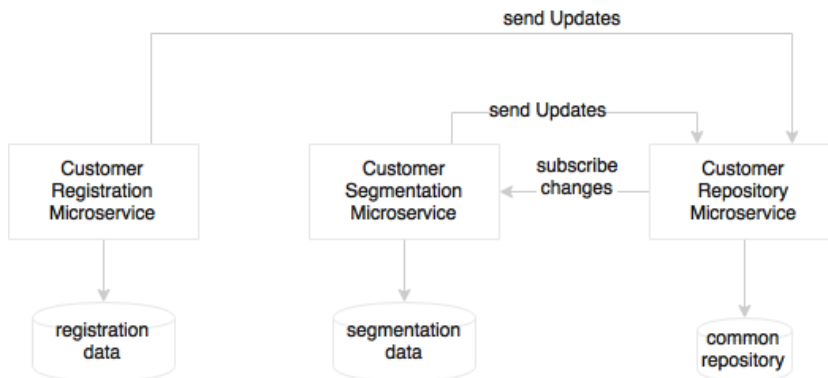


In the preceding diagram, both **Product** and **Order** microservices share one database and one data model. Shared data models, shared schema, and shared tables are recipes for disasters when developing microservices. This may be good at the beginning, but when developing complex microservices, we tend to add relationships between data models, add join queries, and so on. This can result in tightly coupled physical data models.

If the services have only a few tables, it may not be worth investing a full instance of a database like an Oracle database instance. In such cases, a schema level segregation is good enough to start with:



There could be scenarios where we tend to think of using a shared database for multiple services. Taking an example of a customer data repository or master data managed at the enterprise level, the customer registration and customer segmentation microservices logically share the same customer data repository:



As shown in the preceding diagram, an alternate approach in this scenario is to separate the transactional data store for microservices from the enterprise data repository by adding a local transactional data store for these services. This will help the services to have flexibility in remodeling the local data store optimized for its purpose. The enterprise customer repository sends change events when there is any change in the customer data repository. Similarly, if there is any change in any of the transactional data stores, the changes have to be sent to the central customer repository.

Setting up transaction boundaries

Transactions in operational systems are used to maintain the consistency of data stored in an RDBMS by grouping a number of operations together into one atomic block. They either commit or rollback the entire operation. Distributed systems follow the concept of distributed transactions with a two-phase commit. This is particularly required if heterogeneous components such as an RPC service, JMS, and so on participate in a transaction.

Is there a place for transactions in microservices? Transactions are not bad, but one should use transactions carefully, by analyzing what we are trying to do.

For a given microservice, an RDBMS like MySQL may be selected as a backing store to ensure 100% data integrity, for example, a stock or inventory management service where data integrity is key. It is appropriate to define transaction boundaries within the microsystem using local transactions. However, distributed global transactions should be avoided in the microservices context. Proper dependency analysis is required to ensure that transaction boundaries do not span across two different microservices as much as possible.

Altering use cases to simplify transactional requirements

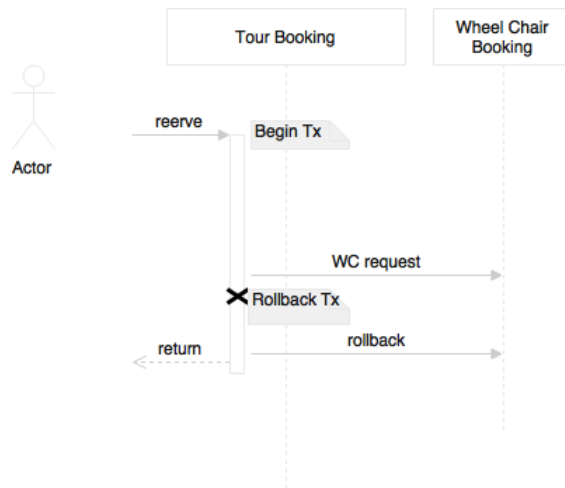
Eventual consistency is a better option than distributed transactions that span across multiple microservices. Eventual consistency reduces a lot of overheads, but application developers may need to re-think the way they write application code. This could include remodeling functions, sequencing operations to minimize failures, batching insert and modify operations, remodeling data structure, and finally, compensating operations that negate the effect

A classical problem is that of the last room selling scenario in a hotel booking use case. What if there is only one room left, and there are multiple customers booking this single available room? A business model change sometimes makes this scenario less impactful. We could set an "under booking profile", where the actual number of bookable rooms can go below the actual number of rooms ($bookable = available - 3$) in anticipation of some cancellations. Anything in this range will be accepted as "subject to confirmation", and customers will be charged only if payment is confirmed. Bookings will be confirmed in a set time window

Now consider the scenario where we are creating customer profiles in a NoSQL database like CouchDB. In more traditional approaches with RDBMS, we insert a customer first, and then insert the customer's address, profile details, the preferences, all in one transaction. When using NoSQL, we may not do the same steps. Instead, we may prepare a JSON object with all the details, and insert this into CouchDB in one go. In this second case, no explicit transaction boundaries are required.

Distributed transaction scenarios

The ideal scenario is to use local transactions within a microservice if required, and completely avoid distributed transactions. There could be scenarios where at the end of the execution of one service, we may want to send a message to another microservice. For example, say a tour reservation has a wheelchair request. Once the reservation is successful, we will have to send a message for the wheelchair booking to another microservice that handles ancillary bookings. The reservation request itself will run on a local transaction. If sending this message fails, we are still in the transaction boundary, and we can roll back the entire transaction. What if we create a reservation and send the message, but after sending the message, we encounter an error in the reservation, the reservation transaction fails, and subsequently, the reservation record is rolled back? Now we end up in a situation where we've unnecessarily created an orphan wheelchair booking:



There are a couple of ways we can address this scenario. The first approach is to delay sending the message till the end. This ensures that there are less chances for any failure after sending the message. Still, if failure occurs after sending the message, then the exception handling routine is run, that is, we send a compensating message to reverse the wheelchair booking.

Service endpoint design consideration

One of the important aspects of microservices is service design. Service design has two key elements: contract design and protocol selection.

Contract design

The first and foremost principle of service design is simplicity. The services should be designed for consumers to consume. A complex service contract reduces the usability of the service. The **KISS (Keep It Simple Stupid)** principle helps us to build better quality services faster, and reduces the cost of maintenance and replacement. The **YAGNI (You Ain't Gonna Need It)** is another principle supporting this idea. Predicting future requirements and building systems are, in reality, not future-proofed. This results in large upfront investment as well as higher cost of maintenance.

Evolutionary design is a great concept. Do just enough design to satisfy today's wants, and keep changing and refactoring the design to accommodate new features as and when they are required. Having said that, this may not be simple unless there is a strong governance in place.

Consumer Driven Contracts (CDC) is a great idea that supports evolutionary design. In many cases, when the service contract gets changed, all consuming applications have to undergo testing. This makes change difficult. CDC helps in building confidence in consumer applications. CDC advocates each consumer to provide their expectation to the provider in the form of test cases so that the provider uses them as integration tests whenever the service contract is changed.

Postel's law is also relevant in this scenario. Postel's law primarily addresses TCP communications; however, this is also equally applicable to service design. When it comes to service design, service providers should be as flexible as possible when accepting consumer requests, whereas service consumers should stick to the contract as agreed with the provider.

Protocol selection

In the SOA world, HTTP/SOAP, and messaging were kinds of default service protocols for service interactions. Microservices follow the same design principles for service interaction. Loose coupling is one of the core principles in the microservices world too.

Microservices fragment applications into many physically independent deployable services. This not only increases the communication cost, it is also susceptible to network failures. This could also result in poor performance of services.

Message-oriented services

If we choose an asynchronous style of communication, the user is disconnected, and therefore, response times are not directly impacted. In such cases, we may use standard JMS or AMQP protocols for communication with JSON as payload. Messaging over HTTP is also popular, as it reduces complexity. Many new entrants in messaging services support HTTP-based communication. Asynchronous REST is also possible, and is handy when calling long-running services.

HTTP and REST endpoints

Communication over HTTP is always better for interoperability, protocol handling, traffic routing, load balancing, security systems, and the like. Since HTTP is stateless, it is more compatible for handling stateless services with no affinity. Most of the development frameworks, testing tools, runtime containers, security systems, and so on are friendlier towards HTTP.

With the popularity and acceptance of REST and JSON, it is the default choice for microservice developers. The HTTP/REST/JSON protocol stack makes building interoperable systems very easy and friendly. HATEOAS is one of the design patterns emerging for designing progressive rendering and self-service navigations. As discussed in the previous chapter, HATEOAS provides a mechanism to link resources together so that the consumer can navigate between resources. RFC 5988 – Web Linking is another upcoming standard.

Optimized communication protocols

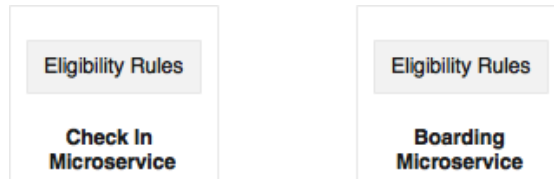
If the service response times are stringent, then we need to pay special attention to the communication aspects. In such cases, we may choose alternate protocols such as Avro, Protocol Buffers, or Thrift for communicating between services. But this limits the interoperability of services. The trade-off is between performance and interoperability requirements. Custom binary protocols need careful evaluation as they bind native objects on both sides – consumer and producer. This could run into release management issues such as class version mismatch in Java-based RPC style communications.

API documentations

Last thing: a good API is not only simple, but should also have enough documentation for the consumers. There are many tools available today for documenting REST-based services like Swagger, RAML, and API Blueprint.

Handling shared libraries

The principle behind microservices is that they should be autonomous and self-contained. In order to adhere to this principle, there may be situations where we will have to duplicate code and libraries. These could be either technical libraries or functional components.

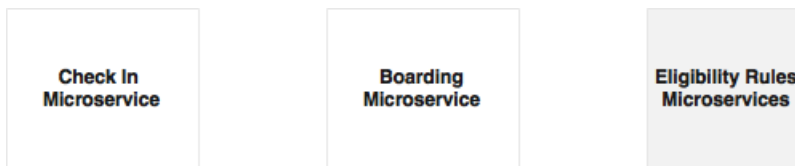


For example, the eligibility for a flight upgrade will be checked at the time of check-in as well as when boarding. If check-in and boarding are two different microservices, we may have to duplicate the eligibility rules in both the services. This was the trade-off between adding a dependency versus code duplication.

It may be easy to embed code as compared to adding an additional dependency, as it enables better release management and performance. But this is against the DRY principle.

DRY principle
 Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

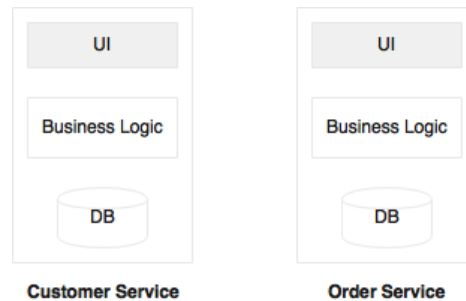
The downside of this approach is that in case of a bug or an enhancement on the shared library, it has to be upgraded in more than one place. This may not be a severe setback as each service can contain a different version of the shared library:



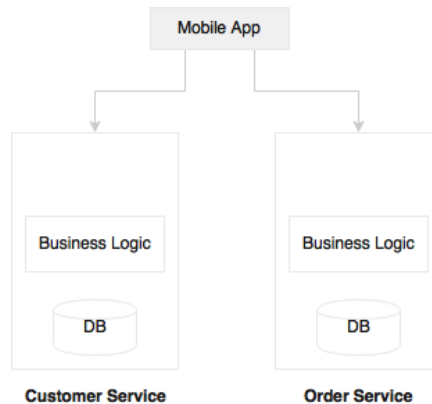
An alternative option of developing the shared library as another microservice itself needs careful analysis. If it is not qualified as a microservice from the business capability point of view, then it may add more complexity than its usefulness. The trade-off analysis is between overheads in communication versus duplicating the libraries in multiple services.

User interfaces in microservices

The microservices principle advocates a microservice as a vertical slice from the database to presentation:

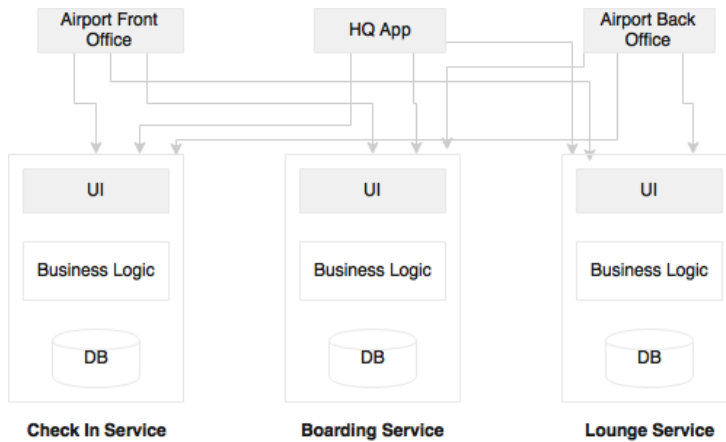


In reality, we get requirements to build quick UI and mobile applications mashing up the existing APIs. This is not uncommon in the modern scenario, where a business wants quick turnaround time from IT:



Penetration of mobile applications is one of the causes of this approach. In many organizations, there will be mobile development teams sitting close to the business team, developing rapid mobile applications by combining and mashing up APIs from multiple sources, both internal and external. In such situations, we may just expose services, and leave it for the mobile teams to realize in the way the business wants. In this case, we will build headless microservices, and leave it to the mobile teams to build a presentation layer.

Another category of problem is that the business may want to build consolidated web applications targeted to communities:



For example, the business may want to develop a departure control application targeting airport users. A departure control web application may have functions such as check-in, lounge management, boarding, and so on. These may be designed as independent microservices. But from the business standpoint, it all needs to be clubbed into a single web application. In such cases, we will have to build web applications by mashing up services from the backend.

One approach is to build a container web application or a placeholder web application, which links to multiple microservices at the backend. In this case, we develop full stack microservices, but the screens coming out of this could be embedded in to another placeholder web application. One of the advantages of this approach is that you can have multiple placeholder web applications targeting different user communities, as shown in the preceding diagram. We may use an API gateway to avoid those crisscross connections. We will explore the API gateway in the next section.

Use of API gateways in microservices

With the advancement of client-side JavaScript frameworks like AngularJS, the server is expected to expose RESTful services. This could lead to two issues. The first issue is the mismatch in contract expectations. The second issue is multiple calls to the server to render a page.

We start with the contract mismatch case. For example, `GetCustomer` may return a JSON with many fields

```
Customer {  
  Name:  
  Address:  
  Contact:  
}
```

In the preceding case, `Name`, `Address`, and `Contact` are nested JSON objects. But a mobile client may expect only basic information such as first name, and last name. In the SOA world, an ESB or a mobile middleware did this job of transformation of data for the client. The default approach in microservices is to get all the elements of `Customer`, and then the client takes up the responsibility to filter the elements. In this case, the overhead is on the network.

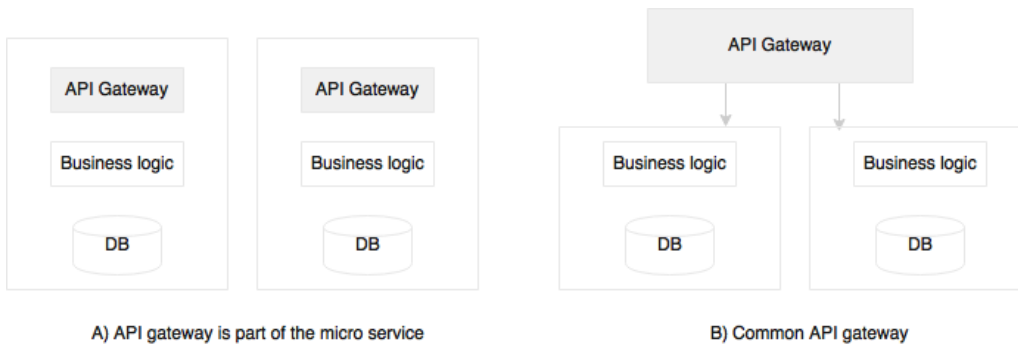
There are several approaches we can think about to solve this case:

```
Customer {  
  Id: 1  
  Name: /customer/name/1  
  Address: /customer/address/1  
  Contact: /customer/contact/1  
}
```

In the first approach, minimal information is sent with links as explained in the section on HATEOAS. In the preceding case, for customer ID 1, there are three links, which will help the client to access specific data elements. The example is a simple logical representation, not the actual JSON. The mobile client in this case will get basic customer information. The client further uses the links to get the additional required information.

The second approach is used when the client makes the REST call; it also sends the required fields as part of the query string. In this scenario, the client sends a request with `firstname` and `lastname` as the query string to indicate that the client only requires these two fields. The downside is that it ends up in complex server-side logic as it has to filter based on the fields. The server has to send different element based on the incoming query.

The third approach is to introduce a level of indirection. In this, a gateway component sits between the client and the server, and transforms data as per the consumer's specification. This is a better approach as we do not compromise on the backend service contract. This leads to what is called UI services. In many cases, the API gateway acts as a proxy to the backend, exposing a set of consumer-specific APIs



There are two ways we can deploy an API gateway. The first one is one API gateway per microservice as shown in diagram **A**. The second approach (diagram **B**) is to have a common API gateway for multiple services. The choice really depends on what we are looking for. If we are using an API gateway as a reverse proxy, then off-the-shelf gateways such as Apigee, Mashery, and the like could be used as a shared platform. If we need fine-grained control over traffic shaping and complex transformations, then per service custom API gateways may be more useful.

A related problem is that we will have to make many calls from the client to the server. If we refer to our holiday example in *Chapter 1, Demystifying Microservices*, you know that for rendering each widget, we had to make a call to the server. Though we transfer only data, it can still add a significant overhead on the network. This approach is not fully wrong, as in many cases, we use responsive design and progressive design. The data will be loaded on demand, based on user navigations. In order to do this, each widget in the client should make independent calls to the server in a lazy mode. If bandwidth is an issue, then an API gateway is the solution. An API gateway acts as a middleman to compose and transform APIs from multiple microservices.

Use of ESB and iPaaS with microservices

Theoretically, SOA is not all about ESBs, but the reality is that ESBs have always been at the center of many SOA implementations. What would be the role of an ESB in the microservices world?

In general, microservices are fully cloud native systems with smaller footprints. The lightweight characteristics of microservices enable automation of deployments, scaling, and so on. On the contrary, enterprise ESBs are heavyweight in nature, and most of the commercial ESBs are not cloud friendly. The key features of an ESB are protocol mediation, transformation, orchestration, and application adaptors. In a typical microservices ecosystem, we may not need any of these features.

The limited ESB capabilities that are relevant for microservices are already available with more lightweight tools such as an API gateway. Orchestration is moved from the central bus to the microservices themselves. Therefore, there is no centralized orchestration capability expected in the case of microservices. Since the services are set up to accept more universal message exchange styles using REST/JSON calls, no protocol mediation is required. The last piece of capability that we get from ESBs are the adaptors to connect back to the legacy systems. In the case of microservices, the service itself provides a concrete implementation, and hence, there are no legacy connectors required. For these reasons, there is no natural space for ESBs in the microservices world.

Many organizations established ESBs as the backbone for their application integrations (EAI). Enterprise architecture policies in such organizations are built around ESBs. There could be a number of enterprise-level policies such as auditing, logging, security, validation, and so on that would have been in place when integrating using ESB. Microservices, however, advocate a more decentralized governance. ESBs will be an overkill if integrated with microservices.

Not all services are microservices. Enterprises have legacy applications, vendor applications, and so on. Legacy services use ESBs to connect with microservices. ESBs still hold their place for legacy integration and vendor applications to integrate at the enterprise level.

With the advancement of clouds, the capabilities of ESBs are not sufficient to manage integration between clouds, cloud to on-premise, and so on. **Integration Platform as a Service (iPaaS)** is evolving as the next generation application integration platform, which further reduces the role of ESBs. In typical deployments, iPaaS invokes API gateways to access microservices.

Service versioning considerations

When we allow services to evolve, one of the important aspect to consider is service versioning. Service versioning should be considered upfront, and not as an afterthought. Versioning helps us to release new services without breaking the existing consumers. Both the old version and the new version will be deployed side by side.

Semantic versions are widely used for service versioning. A semantic version has three components: **major**, **minor**, and **patch**. Major is used when there is a breaking change, minor is used when there is a backward compatible change, and patch is used when there is a backward compatible bug fix

Versioning could get complicated when there is more than one service in a microservice. It is always simple to version services at the service level compared to the operations level. If there is a change in one of the operations, the service is upgraded and deployed to V2. The version change is applicable to all operations in the service. This is the notion of immutable services.

There are three different ways in which we can version REST services:

- URI versioning
- Media type versioning
- Custom header

In URI versioning, the version number is included in the URL itself. In this case, we just need to be worried about the major versions only. Hence, if there is a minor version change or a patch, the consumers do not need to worry about the changes. It is a good practice to alias the latest version to a non-versioned URI, which is done as follows:

```
/api/v3/customer/1234
/api/customer/1234 - aliased to v3.

@RestController("CustomerControllerV3")
@RequestMapping("api/v3/customer")
public class CustomerController {

}
```

A slightly different approach is to use the version number as part of the URL parameter:

```
api/customer/100?v=1.5
```

In case of media type versioning, the version is set by the client on the HTTP Accept header as follows:

```
Accept: application/vnd.company.customer-v3+json
```

A less effective approach for versioning is to set the version in the custom header:

```
@RequestMapping(value =("/{id}", method = RequestMethod.GET, headers =
{"version=3"})
public Customer getCustomer(@PathVariable("id") long id) {
    //other code goes here.
}
```


In the URI approach, it is simple for the clients to consume services. But this has some inherent issues such as the fact that versioning-nested URI resources could be complex. Indeed, migrating clients is slightly complex as compared to media type approaches, with caching issues for multiple versions of the services, and others. However, these issues are not significant enough for us to not go with a URI approach. Most of the big Internet players such as Google, Twitter, LinkedIn, and Salesforce are following the URI approach.

Design for cross origin

With microservices, there is no guarantee that the services will run from the same host or same domain. Composite UI web applications may call multiple microservices for accomplishing a task, and these could come from different domains and hosts.

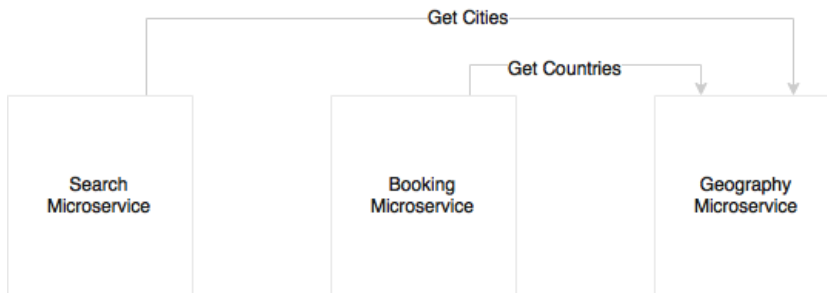
CORS allows browser clients to send requests to services hosted on different domains. This is essential in a microservices-based architecture.

One approach is to enable all microservices to allow cross origin requests from other trusted domains. The second approach is to use an API gateway as a single trusted domain for the clients.

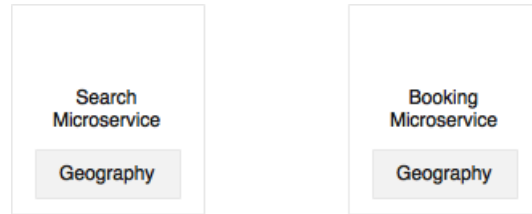
Handling shared reference data

When breaking large applications, one of the common issues which we see is the management of master data or reference data. Reference data is more like shared data required between different microservices. City master, country master, and so on will be used in many services such as flight schedules, reservations, and others

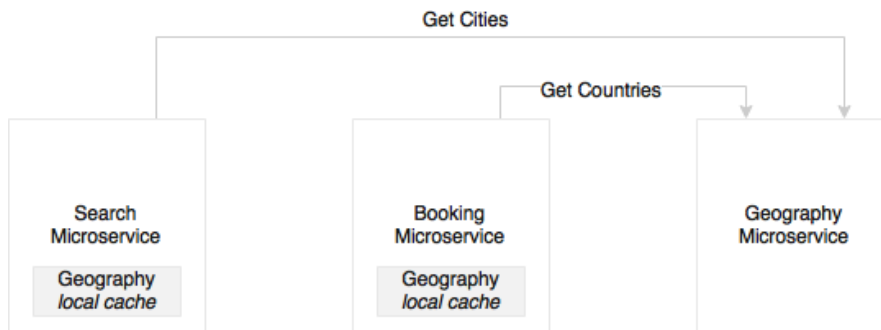
There are a few ways in which we can solve this. For instance, in the case of relatively static, never changing data, then every service can hardcode this data within all the microservices themselves:



Another approach, as shown in the preceding diagram, is to build it as another microservice. This is good, clean, and neat, but the downside is that every service may need to call the master data multiple times. As shown in the diagram for the **Search** and **Booking** example, there are transactional microservices, which use the **Geography** microservice to access shared data:



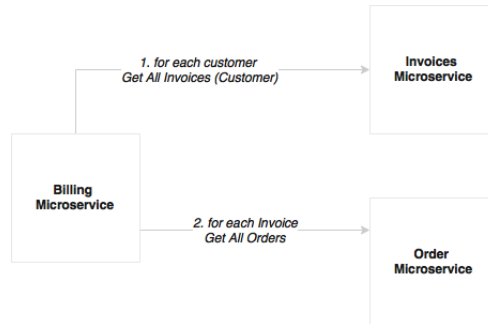
Another option is to replicate the data with every microservice. There is no single owner, but each service has its required master data. When there is an update, all the services are updated. This is extremely performance friendly, but one has to duplicate the code in all the services. It is also complex to keep data in sync across all microservices. This approach makes sense if the code base and data is simple or the data is more static.



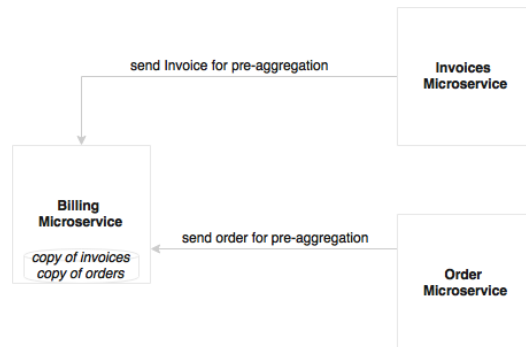
Yet another approach is similar to the first approach, but each service has a local near cache of the required data, which will be loaded incrementally. A local embedded cache such as Ehcache or data grids like Hazelcast or Infinispan could also be used based on the data volumes. This is the most preferred approach for a large number of microservices that have dependency on the master data.

Microservices and bulk operations

Since we have broken monolithic applications into smaller, focused services, it is no longer possible to use join queries across microservice data stores. This could lead to situations where one service may need many records from other services to perform its function.



For example, a monthly billing function needs the invoices of many customers to process the billing. To make it a bit more complicated, invoices may have many orders. When we break billing, invoices, and orders into three different microservices, the challenge that arises is that the **Billing** service has to query the **Invoices** service for each customer to get all the invoices, and then for each invoice, call the **Order** service for getting the orders. This is not a good solution, as the number of calls that goes to other microservices are high:



There are two ways we can think about for solving this. The first approach is to pre-aggregate data as and when it is created. When an order is created, an event is sent out. Upon receiving the event, the **Billing** microservice keeps aggregating data internally for monthly processing. In this case, there is no need for the **Billing** microservice to go out for processing. The downside of this approach is that there is duplication of data.

A second approach, when pre-aggregation is not possible, is to use batch APIs. In such cases, we call `GetAllInvoices`, then we use multiple batches, and each batch further uses parallel threads to get orders. Spring Batch is useful in these situations.

Microservices challenges

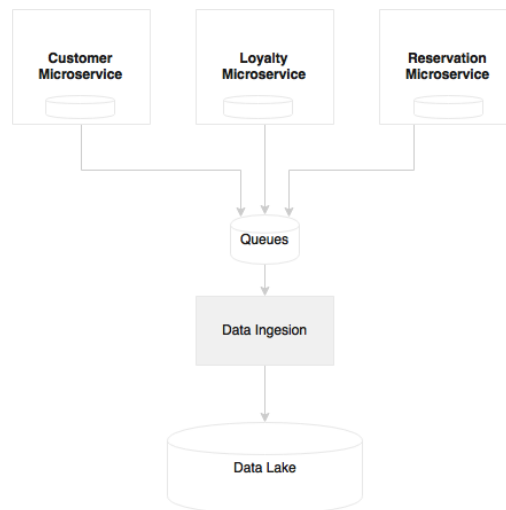
In the previous section, you learned about the right design decisions to be taken, and the trade-offs to be applied. In this section, we will review some of the challenges with microservices, and how to address them for a successful microservice development.

Data islands

Microservices abstract their own local transactional store, which is used for their own transactional purposes. The type of store and the data structure will be optimized for the services offered by the microservice.

For example, if we want to develop a customer relationship graph, we may use a graph database like Neo4j, OrientDB, and the like. A predictive text search to find out a customer based on any related information such as passport number, address, e-mail, phone, and so on could be best realized using an indexed search database like Elasticsearch or Solr.

This will place us into a unique situation of fragmenting data into heterogeneous data islands. For example, Customer, Loyalty Points, Reservations, and others are different microservices, and hence, use different databases. What if we want to do a near real-time analysis of all high value customers by combining data from all three data stores? This was easy with a monolithic application, because all the data was present in a single database:



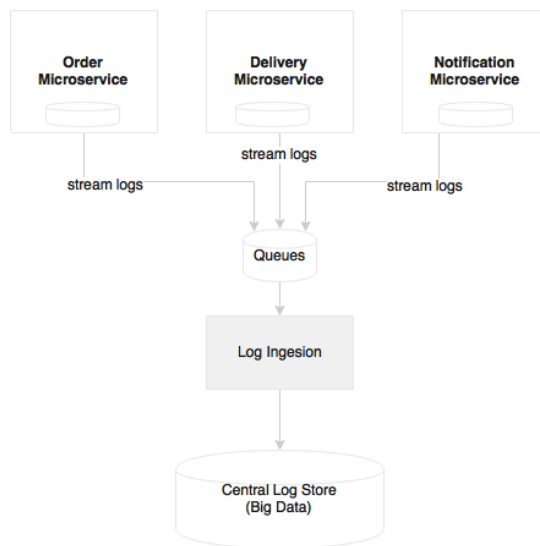
In order to satisfy this requirement, a data warehouse or a data lake is required. Traditional data warehouses like Oracle, Teradata, and others are used primarily for batch reporting. But with NoSQL databases (like Hadoop) and microbatching techniques, near real-time analytics is possible with the concept of data lakes. Unlike the traditional warehouses that are purpose-built for batch reporting, data lakes store raw data without assuming how the data is going to be used. Now the question really is how to port the data from microservices into data lakes.

Data porting from microservices to a data lake or a data warehouse can be done in many ways. Traditional ETL could be one of the options. Since we allow backdoor entry with ETL, and break the abstraction, this is not considered an effective way for data movement. A better approach is to send events from microservices as and when they occur, for example, customer registration, customer update events, and so on. Data ingestion tools consume these events, and propagate the state change to the data lake appropriately. The data ingestion tools are highly scalable platforms such as Spring Cloud Data Flow, Kafka, Flume, and so on.

Logging and monitoring

Log files are a good piece of information for analysis and debugging. Since each microservice is deployed independently, they emit separate logs, maybe to a local disk. This results in fragmented logs. When we scale services across multiple machines, each service instance could produce separate log files. This makes it extremely difficult to debug and understand the behavior of the services through log mining.

Examining **Order**, **Delivery**, and **Notification** as three different microservices, we find no way to correlate a customer transaction that runs across all three of them



When implementing microservices, we need a capability to ship logs from each service to a centrally managed log repository. With this approach, services do not have to rely on the local disk or local I/Os. A second advantage is that the log files are centrally managed, and are available for all sorts of analysis such as historical, real time, and trending. By introducing a correlation ID, end-to-end transactions can be easily tracked.

With a large number of microservices, and with multiple versions and service instances, it would be difficult to find out which service is running on which server what's the health of these services, the service dependencies, and so on. This was much easier with monolithic applications that are tagged against a specific or a fixed set of servers

Apart from understanding the deployment topology and health, it also poses a challenge in identifying service behaviors, debugging, and identifying hotspots. Strong monitoring capabilities are required to manage such an infrastructure.

We will cover the logging and monitoring aspects in *Chapter 7, Logging and Monitoring Microservices*.

Dependency management

Dependency management is one of the key issues in large microservice deployments. How do we identify and reduce the impact of a change? How do we know whether all the dependent services are up and running? How will the service behave if one of the dependent services is not available?

Too many dependencies could raise challenges in microservices. Four important design aspects are stated as follows:

- Reducing dependencies by properly designing service boundaries.
- Reducing impacts by designing dependencies as loosely coupled as possible. Also, designing service interactions through asynchronous communication styles.
- Tackling dependency issues using patterns such as circuit breakers.
- Monitoring dependencies using visual dependency graphs.

Organization culture

One of the biggest challenges in microservices implementation is the organization culture. To harness the speed of delivery of microservices, the organization should adopt Agile development processes, continuous integration, automated QA checks, automated delivery pipelines, automated deployments, and automatic infrastructure provisioning.

Organizations following a waterfall development or heavyweight release management processes with infrequent release cycles are a challenge for microservices development. Insufficient automation is also a challenge for microservices deployment.

In short, Cloud and DevOps are supporting facets of microservice development. These are essential for successful microservices implementation.

Governance challenges

Microservices impose decentralized governance, and this is quite in contrast with the traditional SOA governance. Organizations may find it hard to come up with this change, and that could negatively impact the microservices development.

There are number of challenges that comes with a decentralized governance model. How do we understand who is consuming a service? How do we ensure service reuse? How do we define which services are available in the organization? How do we ensure enforcement of enterprise policies?

The first thing is to have a set of standards, best practices, and guidelines on how to implement better services. These should be available to the organization in the form of standard libraries, tools, and techniques. This ensures that the services developed are top quality, and that they are developed in a consistent manner.

The second important consideration is to have a place where all stakeholders can not only see all the services, but also their documentations, contracts, and service-level agreements. Swagger and API Blueprint are commonly used for handling these requirements.

Operation overheads

Microservices deployment generally increases the number of deployable units and virtual machines (or containers). This adds significant management overheads and increases the cost of operations.

With a single application, a dedicated number of containers or virtual machines in an on-premise data center may not make much sense unless the business benefit is very high. The cost generally goes down with economies of scale. A large number of microservices that are deployed in a shared infrastructure which is fully automated makes more sense, since these microservices are not tagged against any specific VMs or containers. Capabilities around infrastructure automation, provisioning, containerized deployment, and so on are essential for large scale microservices deployments. Without this automation, it would result in a significant operation overhead and increased cost.

With many microservices, the number of **configurable items (CIs)** becomes too high, and the number of servers in which these CIs are deployed might also be unpredictable. This makes it extremely difficult to manage data in a traditional **Configuration Management Database (CMDB)**. In many cases, it is more useful to dynamically discover the current running topology than a statically configured CMDB-style deployment topology.

Testing microservices

Microservices also pose a challenge for the testability of services. In order to achieve a full-service functionality, one service may rely on another service, and that, in turn, on another service—either synchronously or asynchronously. The issue is how do we test an end-to-end service to evaluate its behavior? The dependent services may or may not be available at the time of testing.

Service virtualization or service mocking is one of the techniques used for testing services without actual dependencies. In testing environments, when the services are not available, mock services can simulate the behavior of the actual service. The microservices ecosystem needs service virtualization capabilities. However, this may not give full confidence, as there may be many corner cases that mock services do not simulate, especially when there are deep dependencies.

Another approach, as discussed earlier, is to use a consumer driven contract. The translated integration test cases can cover more or less all corner cases of the service invocation.

Test automation, appropriate performance testing, and continuous delivery approaches such as A/B testing, feature flags, canary testing, blue-green deployments, and red-black deployments, all reduce the risks of production releases.

Infrastructure provisioning

As briefly touched on under operation overheads, manual deployment could severely challenge the microservices rollouts. If a deployment has manual elements, the deployer or operational administrators should know the running topology, manually reroute traffic, and then deploy the application one by one till all services are upgraded. With many server instances running, this could lead to significant operational overheads. Moreover, the chances of errors are high in this manual approach.

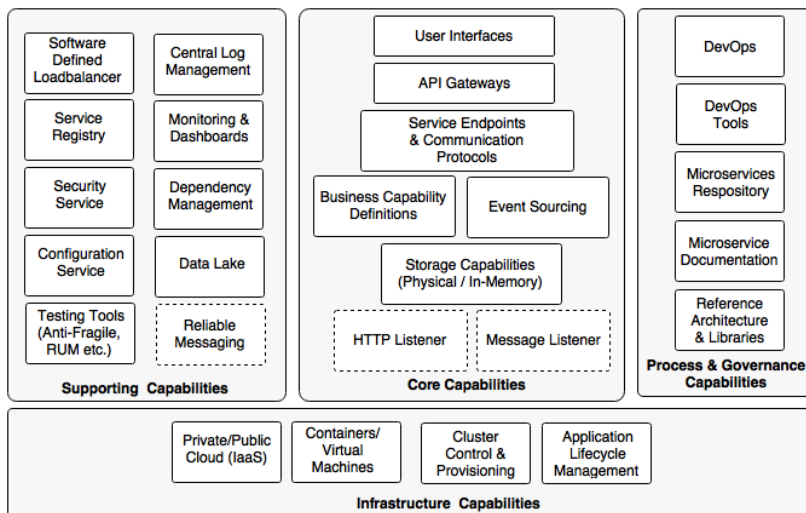
Microservices require a supporting elastic cloud-like infrastructure which can automatically provision VMs or containers, automatically deploy applications, adjust traffic flows, replicate new version to all instances, and gracefully phase out older versions. The automation also takes care of scaling up elastically by adding containers or VMs on demand, and scaling down when the load falls below threshold.

In a large deployment environment with many microservices, we may also need additional tools to manage VMs or containers that can further initiate or destroy services automatically.

The microservices capability model

Before we conclude this chapter, we will review a capability model for microservices based on the design guidelines and common pattern and solutions described in this chapter.

The following diagram depicts the microservices capability model:



The capability model is broadly classified in to four areas

- **Core capabilities:** These are part of the microservices themselves
- **Supporting capabilities:** These are software solutions supporting core microservice implementations
- **Infrastructure capabilities:** These are infrastructure level expectations for a successful microservices implementation
- **Governance capabilities:** These are more of process, people, and reference information

Core capabilities

The core capabilities are explained as follows:

- **Service listeners** (HTTP/messaging): If microservices are enabled for a HTTP-based service endpoint, then the HTTP listener is embedded within the microservices, thereby eliminating the need to have any external application server requirement. The HTTP listener is started at the time of the application startup. If the microservice is based on asynchronous communication, then instead of an HTTP listener, a message listener is started. Optionally, other protocols could also be considered. There may not be any listeners if the microservice is a scheduled service. Spring Boot and Spring Cloud Streams provide this capability.
- **Storage capability:** The microservices have some kind of storage mechanisms to store state or transactional data pertaining to the business capability. This is optional, depending on the capabilities that are implemented. The storage could be either a physical storage (RDBMS such as MySQL; NoSQL such as Hadoop, Cassandra, Neo 4J, Elasticsearch, and so on), or it could be an in-memory store (cache like Ehcache, data grids like Hazelcast, Infinispan, and so on)
- **Business capability definition:** This is the core of microservices, where the business logic is implemented. This could be implemented in any applicable language such as Java, Scala, Conjure, Erlang, and so on. All required business logic to fulfill the function will be embedded within the microservices themselves.
- **Event sourcing:** Microservices send out state changes to the external world without really worrying about the targeted consumers of these events. These events could be consumed by other microservices, audit services, replication services, or external applications, and the like. This allows other microservices and applications to respond to state changes.

- **Service endpoints and communication protocols:** These define the APIs for external consumers to consume. These could be synchronous endpoints or asynchronous endpoints. Synchronous endpoints could be based on REST/JSON or any other protocols such as Avro, Thrift, Protocol Buffers, and so on. Asynchronous endpoints are through Spring Cloud Streams backed by RabbitMQ, other messaging servers, or other messaging style implementations such as ZeroMQ.
- **API gateway:** The API gateway provides a level of indirection by either proxying service endpoints or composing multiple service endpoints. The API gateway is also useful for policy enforcements. It may also provide real time load balancing capabilities. There are many API gateways available in the market. Spring Cloud Zuul, Mashery, Apigee, and 3scale are some examples of the API gateway providers.
- **User interfaces:** Generally, user interfaces are also part of microservices for users to interact with the business capabilities realized by the microservices. These could be implemented in any technology, and are channel- and device-agnostic.

Infrastructure capabilities

Certain infrastructure capabilities are required for a successful deployment, and managing large scale microservices. When deploying microservices at scale, not having proper infrastructure capabilities can be challenging, and can lead to failures:

- **Cloud:** Microservices implementation is difficult in a traditional data center environment with long lead times to provision infrastructures. Even a large number of infrastructures dedicated per microservice may not be very cost effective. Managing them internally in a data center may increase the cost of ownership and cost of operations. A cloud-like infrastructure is better for microservices deployment.
- **Containers or virtual machines:** Managing large physical machines is not cost effective, and they are also hard to manage. With physical machines, it is also hard to handle automatic fault tolerance. Virtualization is adopted by many organizations because of its ability to provide optimal use of physical resources. It also provides resource isolation. It also reduces the overheads in managing large physical infrastructure components. Containers are the next generation of virtual machines. VMWare, Citrix, and so on provide virtual machine technologies. Docker, Drawbridge, Rocket, and LXD are some of the containerizer technologies.

- **Cluster control and provisioning:** Once we have a large number of containers or virtual machines, it is hard to manage and maintain them automatically. Cluster control tools provide a uniform operating environment on top of the containers, and share the available capacity across multiple services. Apache Mesos and Kubernetes are examples of cluster control systems.
- **Application lifecycle management:** Application lifecycle management tools help to invoke applications when a new container is launched, or kill the application when the container shuts down. Application life cycle management allows for script application deployments and releases. It automatically detects failure scenario, and responds to those failures thereby ensuring the availability of the application. This works in conjunction with the cluster control software. Marathon partially addresses this capability.

Supporting capabilities

Supporting capabilities are not directly linked to microservices, but they are essential for large scale microservices development:

- **Software defined load balancer:** The load balancer should be smart enough to understand the changes in the deployment topology, and respond accordingly. This moves away from the traditional approach of configuring static IP addresses, domain aliases, or cluster addresses in the load balancer. When new servers are added to the environment, it should automatically detect this, and include them in the logical cluster by avoiding any manual interactions. Similarly, if a service instance is unavailable, it should take it out from the load balancer. A combination of Ribbon, Eureka, and Zuul provide this capability in Spring Cloud Netflix.
- **Central log management:** As explored earlier in this chapter, a capability is required to centralize all logs emitted by service instances with the correlation IDs. This helps in debugging, identifying performance bottlenecks, and predictive analysis. The result of this is fed back into the life cycle manager to take corrective actions.
- **Service registry:** A service registry provides a runtime environment for services to automatically publish their availability at runtime. A registry will be a good source of information to understand the services topology at any point. Eureka from Spring Cloud, Zookeeper, and Etcd are some of the service registry tools available.

- **Security service:** A distributed microservices ecosystem requires a central server for managing service security. This includes service authentication and token services. OAuth2-based services are widely used for microservices security. Spring Security and Spring Security OAuth are good candidates for building this capability.
- **Service configuration:** All service configurations should be externalized as discussed in the Twelve-Factor application principles. A central service for all configurations is a good choice. Spring Cloud Config server, and Archaius are out-of-the-box configuration servers.
- **Testing tools (anti-fragile, RUM, and so on):** Netflix uses Simian Army for anti-fragile testing. Matured services need consistent challenges to see the reliability of the services, and how good fallback mechanisms are. Simian Army components create various error scenarios to explore the behavior of the system under failure scenarios.
- **Monitoring and dashboards:** Microservices also require a strong monitoring mechanism. This is not just at the infrastructure-level monitoring but also at the service level. Spring Cloud Netflix Turbine, Hystrix Dashboard, and the like provide service level information. End-to-end monitoring tools like AppDynamic, New Relic, Dynatrace, and other tools like statd, Sensu, and Spigo could add value to microservices monitoring.
- **Dependency and CI management:** We also need tools to discover runtime topologies, service dependencies, and to manage configurable items. A graph-based CMDB is the most obvious tool to manage these scenarios.
- **Data lake:** As discussed earlier in this chapter, we need a mechanism to combine data stored in different microservices, and perform near real-time analytics. A data lake is a good choice for achieving this. Data ingestion tools like Spring Cloud Data Flow, Flume, and Kafka are used to consume data. HDFS, Cassandra, and the like are used for storing data.
- **Reliable messaging:** If the communication is asynchronous, we may need a reliable messaging infrastructure service such as RabbitMQ or any other reliable messaging service. Cloud messaging or messaging as a service is a popular choice in Internet scale message-based service endpoints.

Process and governance capabilities

The last piece in the puzzle is the process and governance capabilities that are required for microservices:

- **DevOps:** The key to successful implementation of microservices is to adopt DevOps. DevOps compliment microservices development by supporting Agile development, high velocity delivery, automation, and better change management.

- **DevOps tools:** DevOps tools for Agile development, continuous integration, continuous delivery, and continuous deployment are essential for successful delivery of microservices. A lot of emphasis is required on automated functioning, real user testing, synthetic testing, integration, release, and performance testing.
- **Microservices repository:** A microservices repository is where the versioned binaries of microservices are placed. These could be a simple Nexus repository or a container repository such as a Docker registry.
- **Microservice documentation:** It is important to have all microservices properly documented. Swagger or API Blueprint are helpful in achieving good microservices documentation.
- **Reference architecture and libraries:** The reference architecture provides a blueprint at the organization level to ensure that the services are developed according to certain standards and guidelines in a consistent manner. Many of these could then be translated to a number of reusable libraries that enforce service development philosophies.

Summary

In this chapter, you learned about handling practical scenarios that will arise in microservices development.

You learned various solution options and patterns that could be applied to solve common microservices problems. We reviewed a number of challenges when developing large scale microservices, and how to address those challenges effectively.

We also built a capability reference model for a microservices-based ecosystem. The capability model helps in addressing gaps when building Internet scale microservices. The capability model learned in this chapter will be the backbone for this book. The remaining chapters will deep dive into the capability model.

In the next chapter, we will take a real-world problem and model it using the microservices architecture to see how to translate our learnings into practice.

4

Microservices Evolution – A Case Study

Like SOA, a microservices architecture can be interpreted differently by different organizations, based on the problem in hand. Unless a sizable, real world problem is examined in detail, microservices concepts are hard to understand.

This chapter will introduce BrownField Airline (BF), a fictitious budget airline, and their journey from a monolithic **Passenger Sales and Service (PSS)** application to a next generation microservices architecture. This chapter examines the PSS application in detail, and explains the challenges, approach, and transformation steps of a monolithic system to a microservices-based architecture, adhering to the principles and practices that were explained in the previous chapter.

The intention of this case study is to get us as close as possible to a live scenario so that the architecture concepts can be set in stone.

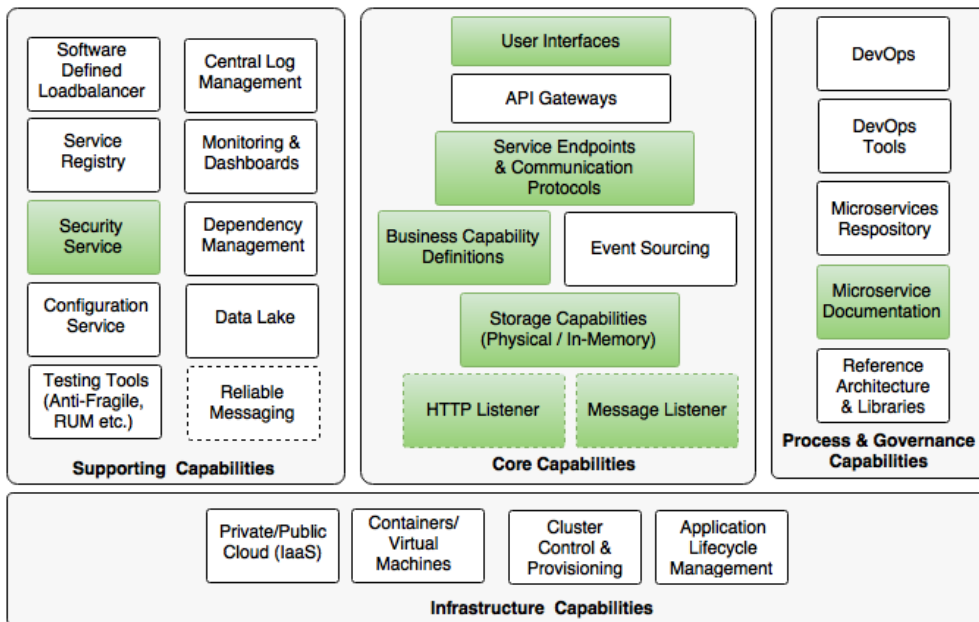
By the end of this chapter, you will have learned about the following:

- A real world case for migrating monolithic systems to microservices-based ones, with the BrownField Airline's PSS application as an example
- Various approaches and transition strategies for migrating a monolithic application to microservices
- Designing a new futuristic microservices system to replace the PSS application using Spring Framework components

Reviewing the microservices capability model

The examples in this chapter explore the following microservices capabilities from the microservices capability model discussed in *Chapter 3, Applying Microservices Concepts*:

- **HTTP Listener**
- **Message Listener**
- **Storage Capabilities (Physical/In-Memory)**
- **Business Capability Definitions**
- **Service Endpoints & Communication Protocols**
- **User Interfaces**
- **Security Service**
- **Microservice Documentation**



In *Chapter 2, Building Microservices with Spring Boot*, we explored all these capabilities in isolation including how to secure Spring Boot microservices. This chapter will build a comprehensive microservices example based on a real world case study.



The full source code of this chapter is available under the Chapter 4 projects in the code files

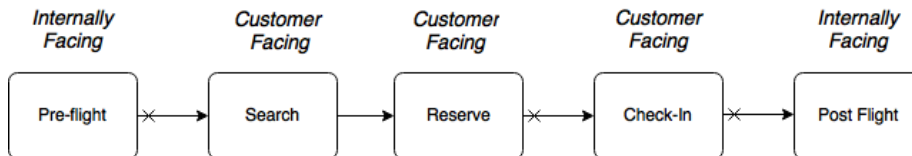


Understanding the PSS application

BrownField Airline is one of the fastest growing low-cost, regional airlines, flying directly to more than 100 destinations from its hub. As a start-up airline, BrownField Airline started its operations with few destinations and few aircrafts. BrownField developed its home-grown PSS application to handle their passenger sales and services.

Business process view

This use case is considerably simplified for discussion purposes. The process view in the following diagram shows BrownField Airline's end-to-end passenger services operations covered by the current PSS solution:

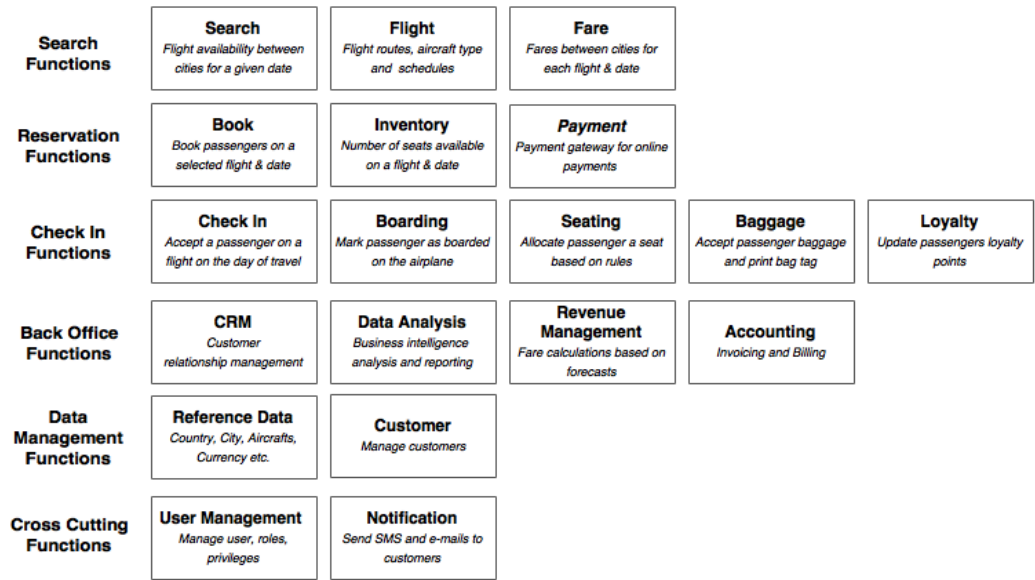


The current solution is automating certain customer-facing functions as well as certain internally facing functions. There are two internally facing functions, **Pre-flight** and **Post-flight**. **Pre-flight** functions include the planning phase, used for preparing flight schedules, plans, aircrafts, and so on. **Post-flight** functions are used by the back office for revenue management, accounting, and so on. The **Search** and **Reserve** functions are part of the online seat reservation process, and the **Check-in** function is the process of accepting passengers at the airport. The **Check-in** function is also accessible to the end users over the Internet for online check-in.

The cross marks at the beginning of the arrows in the preceding diagram indicate that they are disconnected, and occur at different timelines. For example, passengers are allowed to book 360 days in advance, whereas the check-in generally happens 24 hours before flight departure

Functional view

The following diagram shows the functional building blocks of BrownField Airline's PSS landscape. Each business process and its related subfunctions are represented in a row:

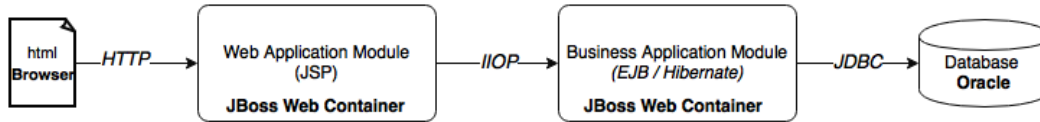


Each subfunction shown in the preceding diagram explains its role in the overall business process. Some subfunctions participate in more than one business process. For example, inventory is used in both search as well as in booking. To avoid any complication, this is not shown in the diagram. Data management and cross-cutting subfunctions are used across many business functions.

Architectural view

In order to effectively manage the end-to-end passenger operations, BrownField had developed an in-house PSS application, almost ten years back. This well-architected application was developed using Java and JEE technologies combined with the best-of-the-breed open source technologies available at the time.

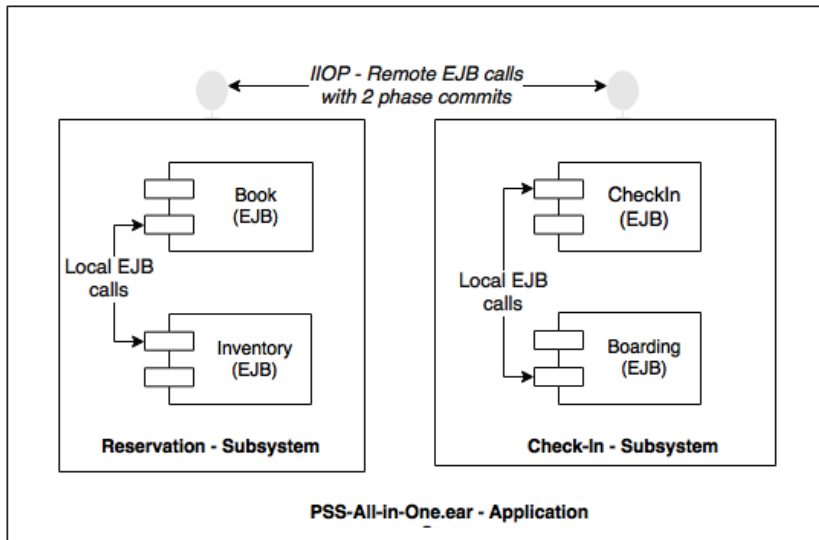
The overall architecture and technologies are shown in the following diagram:



The architecture has well-defined boundaries. Also, different concerns are separated into different layers. The web application was developed as an *N*-tier, component-based modular system. The functions interact with each other through well-defined service contracts defined in the form of EJB endpoints

Design view

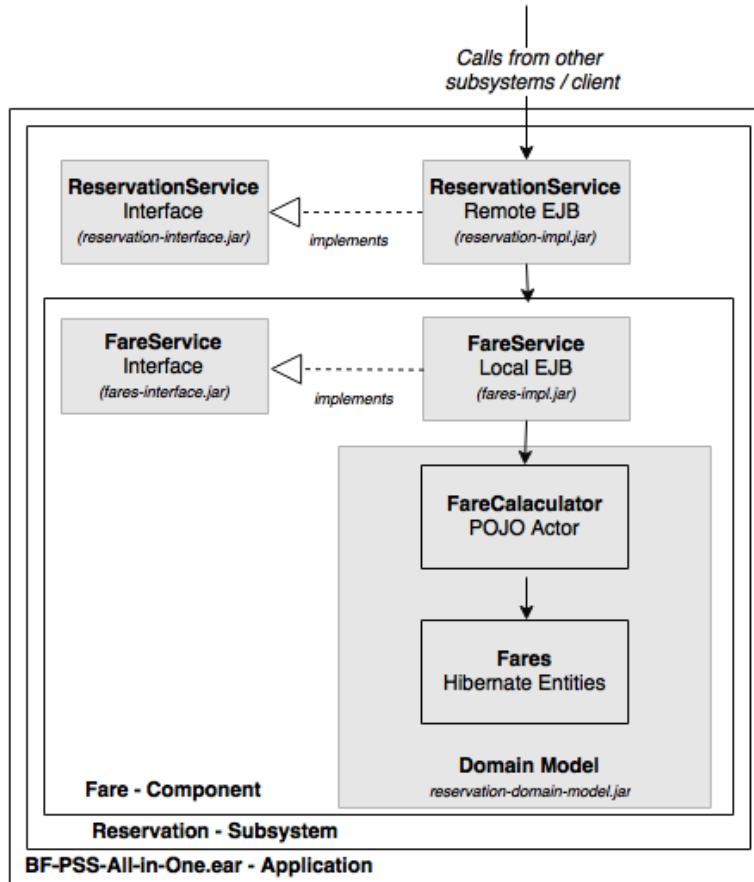
The application has many logical functional groupings or subsystems. Further, each subsystem has many components organized as depicted in the next diagram:



Subsystems interact with each other through remote EJB calls using the IIOP protocol. The transactional boundaries span across subsystems. Components within the subsystems communicate with each other through local EJB component interfaces. In theory, since subsystems use remote EJB endpoints, they could run on different physically separated application servers. This was one of the design goals.

Implementation view

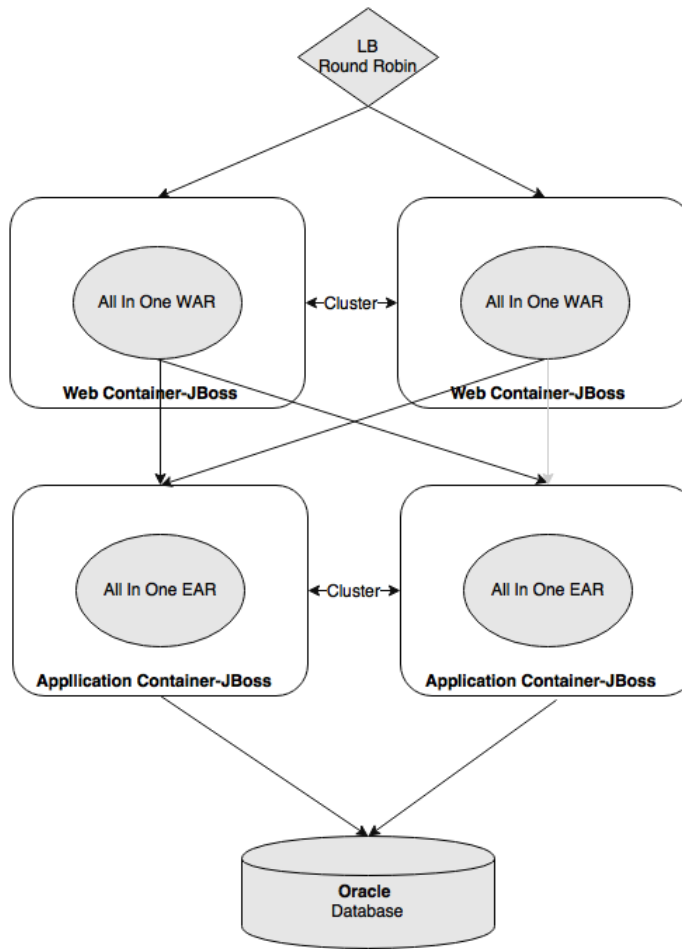
The implementation view in the following diagram showcases the internal organization of a subsystem and its components. The purpose of the diagram is also to show the different types of artifacts:



In the preceding diagram, the gray-shaded boxes are treated as different Maven projects, and translate into physical artifacts. Subsystems and components are designed adhering to the *program to an interface* principle. Interfaces are packaged as separate JAR files so that clients are abstracted from the implementations. The complexity of the business logic is buried in the domain model. Local EJBs are used as component interfaces. Finally, all subsystems are packaged into a single all-in-one EAR, and deployed in the application server.

Deployment view

The application's initial deployment was simple and straightforward as shown in the next diagram:



The web modules and business modules were deployed into separate application server clusters. The application was scaled horizontally by adding more and more application servers to the cluster.

Zero downtime deployments were handled by creating a standby cluster, and gracefully diverting the traffic to that cluster. The standby cluster is destroyed once the primary cluster is patched with the new version and brought back to service. Most of the database changes were designed for backward compatibility, but breaking changes were promoted with application outages.

Death of the monolith

The PSS application was performing well, successfully supporting all business requirements as well as the expected service levels. The system had no issues in scaling with the organic growth of the business in the initial years.

The business has seen tremendous growth over a period of time. The fleet size increased significantly, and new destinations got added to the network. As a result of this rapid growth, the number of bookings has gone up, resulting in a steep increase in transaction volumes, up to 200 - to 500 - fold of what was originally estimated.

Pain points

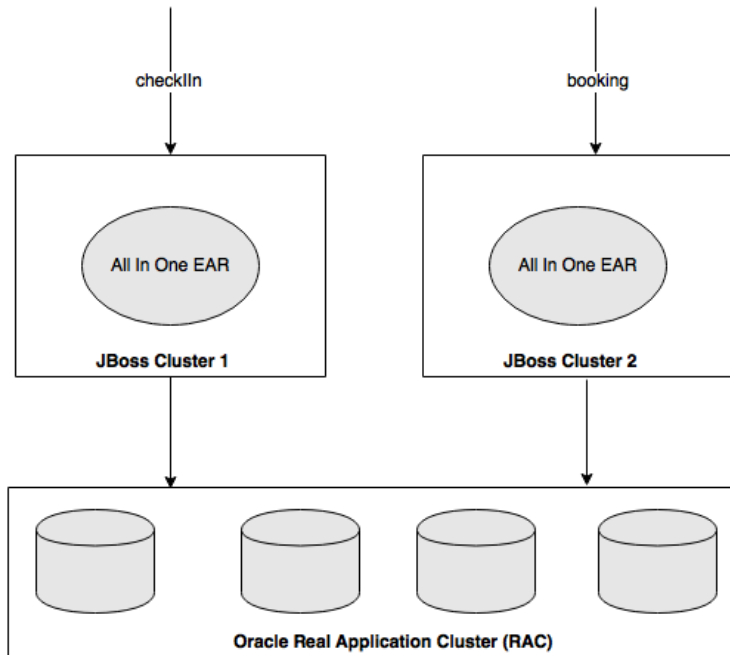
The rapid growth of the business eventually put the application under pressure. Odd stability issues and performance issues surfaced. New application releases started breaking the working code. Moreover, the cost of change and the speed of delivery started impacting the business operations profoundly.

An end-to-end architecture review was ordered, and it exposed the weaknesses of the system as well as the root causes of many failures, which were as follows:

- **Stability:** The stability issues are primarily due to stuck threads, which limit the application server's capability to accept more transactions. The stuck threads are mainly due to database table locks. Memory issues are another contributor to the stability issues. There were also issues in certain resource intensive operations that were impacting the whole application.
- **Outages:** The outage window increased largely because of the increase in server startup time. The root cause of this issue boiled down to the large size of the EAR. Message pile up during any outage windows causes heavy usage of the application immediately after an outage window. Since everything is packaged in a single EAR, any small application code change resulted in full redeployment. The complexity of the zero downtime deployment model described earlier, together with the server startup times increased both the number of outages and their duration.
- **Agility:** The complexity of the code also increased considerably over time, partially due to the lack of discipline in implementing the changes. As a result, changes became harder to implement. Also, the impact analysis became too complex to perform. As a result, inaccurate impact analysis often led to fixes that broke the working code. The application build time went up severely, from a few minutes to hours, causing unacceptable drops in development productivity. The increase in build time also led to difficulty in build automation, and eventually stopped **continuous integration (CI)** and unit testing.

Stop gap fix

Performance issues were partially addressed by applying the Y-axis scale method in the scale cube, as described in *Chapter 1, Demystifying Microservices*. The all-encompassing EAR is deployed into multiple disjoint clusters. A software proxy was installed to selectively route the traffic to designated clusters as shown in the following diagram:



This helped BrownField's IT to scale the application servers. Therefore, the stability issues were controlled. However, this soon resulted in a bottleneck at the database level. Oracle's **Real Application Cluster (RAC)** was implemented as a solution to this problem at the database layer.

This new scaling model reduced the stability issues, but at a premium of increased complexity and cost of ownership. The technology debt also increased over a period of time, leading to a state where a complete rewrite was the only option for reducing this technology debt.

Retrospection

Although the application was well-architected, there was a clear segregation between the functional components. They were loosely coupled, programmed to interfaces, with access through standards-based interfaces, and had a rich domain model.

The obvious question is, how come such a well-architected application failed to live up to the expectations? What else could the architects have done?

It is important to understand what went wrong over a period of time. In the context of this book, it is also important to understand how microservices can avoid the recurrence of these scenarios. We will examine some of these scenarios in the subsequent sections.

Shared data

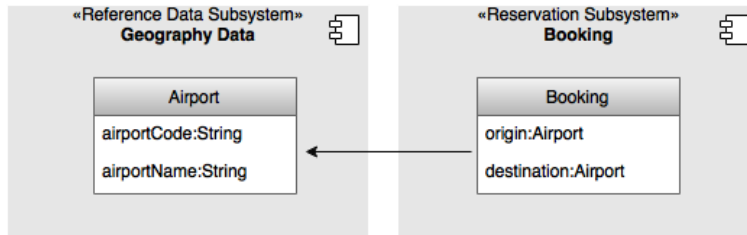
Almost all functional modules require reference data such as the airline's details, airplane details, a list of airports and cities, countries, currencies, and so on. For example, fare is calculated based on the point of origin (city), a flight is between an origin and a destination (airports), check-in is at the origin airport (airport), and so on. In some functions, the reference data is a part of the information model, whereas in some other functions, it is used for validation purposes.

Much of this reference data is neither fully static nor fully dynamic. Addition of a country, city, airport, or the like could happen when the airline introduces new routes. Aircraft reference data could change when the airline purchases a new aircraft, or changes an existing airplane's seat configuration

One of the common usage scenarios of reference data is to filter the operational data based on certain reference data. For instance, say a user wishes to see all the flights to a country. In this case, the flow of events could be as follows: find all the cities in the selected country, then all airports in the cities, and then fire a request to get all the flights to the list of resulting airports identified in that country

The architects considered multiple approaches when designing the system. Separating the reference data as an independent subsystem like other subsystems was one of the options considered, but this could lead to performance issues. The team took the decision to follow an exception approach for handling reference data as compared to other transactions. Considering the nature of the query patterns discussed earlier, the approach was to use the reference data as a shared library.

In this case, the subsystems were allowed to access the reference data directly using pass-by-reference semantic data instead of going through the EJB interfaces. This also meant that irrespective of the subsystems, hibernate entities could use the reference data as a part of their entity relationships:



As depicted in the preceding diagram, the **Booking** entity in the reservation subsystem is allowed to use the reference data entities, in this case **Airport**, as part of their relationships.

Single database

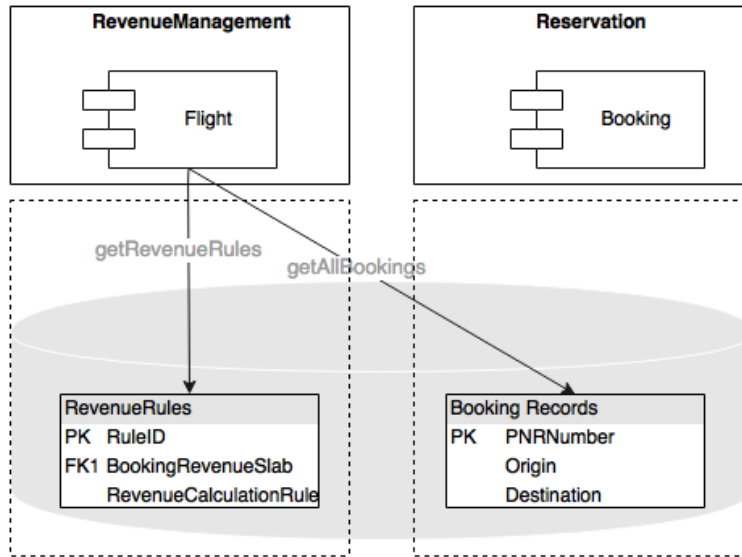
Though enough segregation was enforced at the middle tier, all functions pointed to a single database, even to the same database schema. The single schema approach opened a plethora of issues.

Native queries

The Hibernate framework provides a good abstraction over the underlying databases. It generates efficient SQL statements, in most of the cases targeting the database using specific dialects. However, sometimes, writing native JDBC SQLs offers better performance and resource efficiency. In some cases, using native database functions gives an even better performance.

The single database approach worked well at the beginning. But over a period of time, it opened up a loophole for the developers by connecting database tables owned by different subsystems. Native JDBC SQL was a good vehicle for doing this.

The following diagram shows an example of connecting two tables owned by two subsystems using a native JDBC SQL:



As shown in the preceding diagram, the Accounting component requires all booking records for a day, for a given city, from the Booking component to process the day-end billing. The subsystem-based design enforces Accounting to make a service call to Booking to get all booking records for a given city. Assume this results in N booking records. Now, for each booking record, Accounting has to execute a database call to find the applicable rules based on the fare code attached to each booking record. This could result in $N+1$ JDBC calls, which is inefficient. Workarounds, such as batch queries or parallel and batch executions, are available, but this would lead to increased coding efforts and higher complexity. The developers tackled this issue with a native JDBC query as an easy-to-implement shortcut. Essentially, this approach could reduce the number of calls from $N+1$ to a single database call, with minimal coding efforts.

This habit continued with many JDBC native queries connecting tables across multiple components and subsystems. This resulted not only in tightly coupled components, but also led to undocumented, hard-to-detect code.

Stored procedures

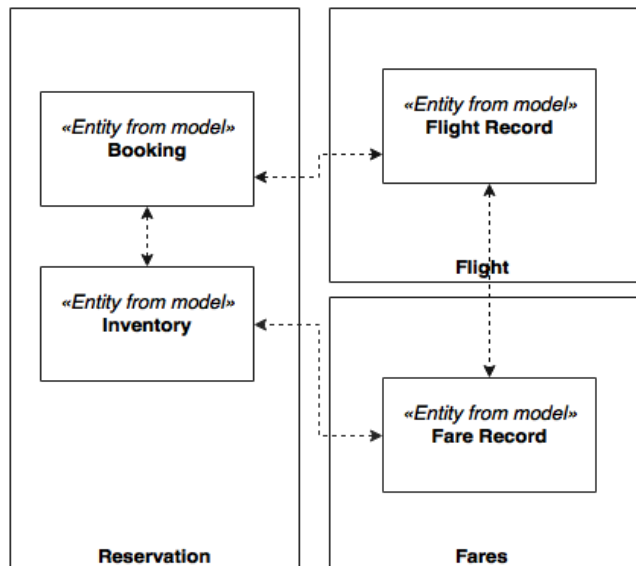
Another issue that surfaced as a result of the use of a single database was the use of complex stored procedures. Some of the complex data-centric logic written at the middle layer was not performing well, causing slow response, memory issues, and thread-blocking issues.

In order to address this problem, the developers took the decision to move some of the complex business logic from the middle tier to the database tier by implementing the logic directly within the stored procedures. This decision resulted in better performance of some of the transactions, and removed some of the stability issues. More and more procedures were added over a period of time. However, this eventually broke the application's modularity.

Domain boundaries

Though the domain boundaries were well established, all the components were packaged as a single EAR file. Since all the components were set to run on a single container, there was no stopping the developers referencing objects across these boundaries. Over a period of time, the project teams changed, delivery pressure increased, and the complexity grew tremendously. The developers started looking for quick solutions rather than the right ones. Slowly, but steadily, the modular nature of the application went away.

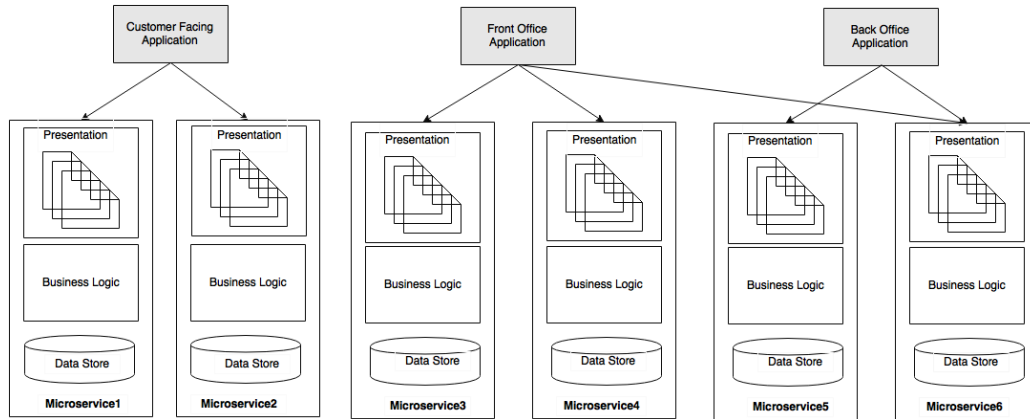
As depicted in the following diagram, hibernate relationships were created across subsystem boundaries:



Microservices to the rescue

There are not many improvement opportunities left to support the growing demand of BrownField Airline's business. BrownField Airline was looking to re-platform the system with an evolutionary approach rather than a revolutionary model.

Microservices is an ideal choice in these situations – for transforming a legacy monolithic application with minimal disruption to the business:



As shown in the preceding diagram, the objective is to move to a microservices-based architecture aligned to the business capabilities. Each microservice will hold the data store, the business logic, and the presentation layer.

The approach taken by BrownField Airline is to build a number of web portal applications targeting specific user communities such as customer facing, front office, and back office. The advantage of this approach lies in the flexibility for modeling and also in the possibility to treat different communities differently. For example, the policies, architecture, and testing approaches for the Internet facing layer are different from the intranet-facing web application. Internet-facing applications may take advantage of **CDNs (Content Delivery Networks)** to move pages as close to the customer as possible, whereas intranet applications could serve pages directly from the data center.

The business case

When building business cases for migration, one of the commonly asked questions is "how does the microservices architecture avoid resurfacing of the same issues in another five years' time?"

Microservices offers a full list of benefits, which you learned in *Chapter 1, Demystifying Microservices*, but it is important to list a few here that are critical in this situation:

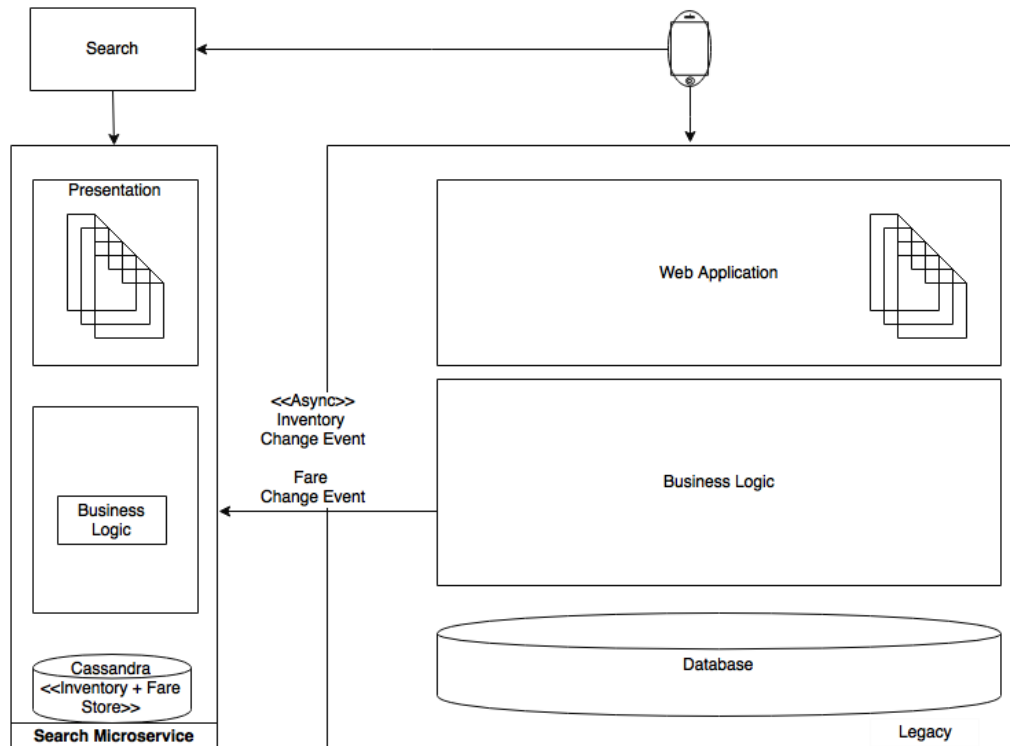
- **Service dependencies:** While migrating from monolithic applications to microservices, the dependencies are better known, and therefore the architects and developers are much better placed to avoid breaking dependencies and to future-proof dependency issues. Lessons from the monolithic application helps architects and developers to design a better system.
- **Physical boundaries:** Microservices enforce physical boundaries in all areas including the data store, the business logic, and the presentation layer. Access across subsystems or microservices are truly restricted due to their physical isolation. Beyond the physical boundaries, they could even run on different technologies.
- **Selective scaling:** Selective scale out is possible in microservices architecture. This provides a much more cost-effective scaling mechanism compared to the Y-scale approach used in the monolithic scenario.
- **Technology obsolescence:** Technology migrations could be applied at a microservices level rather than at the overall application level. Therefore, it does not require a humongous investment.

Plan the evolution

It is not simple to break an application that has millions of lines of code, especially if the code has complex dependencies. How do we break it? More importantly, where do we start, and how do we approach this problem?

Evolutionary approach

The best way to address this problem is to establish a transition plan, and gradually migrate the functions as microservices. At every step, a microservice will be created outside of the monolithic application, and traffic will be diverted to the new service as shown in the following diagram:



In order to run this migration successfully, a number of key questions need to be answered from the transition point of view:

- Identification of microservices' boundaries
- Prioritizing microservices for migration
- Handling data synchronization during the transition phase
- Handling user interface integration, working with old and new user interfaces

- Handling of reference data in the new system
- Testing strategy to ensure the business capabilities are intact and correctly reproduced
- Identification of any prerequisites for microservice development such as microservices capabilities, frameworks, processes, and so on

Identification of microservices boundaries

The first and foremost activity is to identify the microservices' boundaries. This is the most interesting part of the problem, and the most difficult part as well. If identification of the boundaries is not done properly, the migration could lead to more complex manageability issues.

Like in SOA, a service decomposition is the best way to identify services. However, it is important to note that decomposition stops at a business capability or bounded context. In SOA, service decomposition goes further into an atomic, granular service level.

A top-down approach is typically used for domain decomposition. The bottom-up approach is also useful in the case of breaking an existing system, as it can utilize a lot of practical knowledge, functions, and behaviors of the existing monolithic application.

The previous decomposition step will give a potential list of microservices. It is important to note that this isn't the final list of microservices, but it serves as a good starting point. We will run through a number of filtering mechanisms to get to a final list. The first cut of functional decomposition will, in this case, be similar to the diagram shown under the functional view introduced earlier in this chapter.

Analyze dependencies

The next step is to analyze the dependencies between the initial set of candidate microservices that we created in the previous section. At the end of this activity, a dependency graph will be produced.

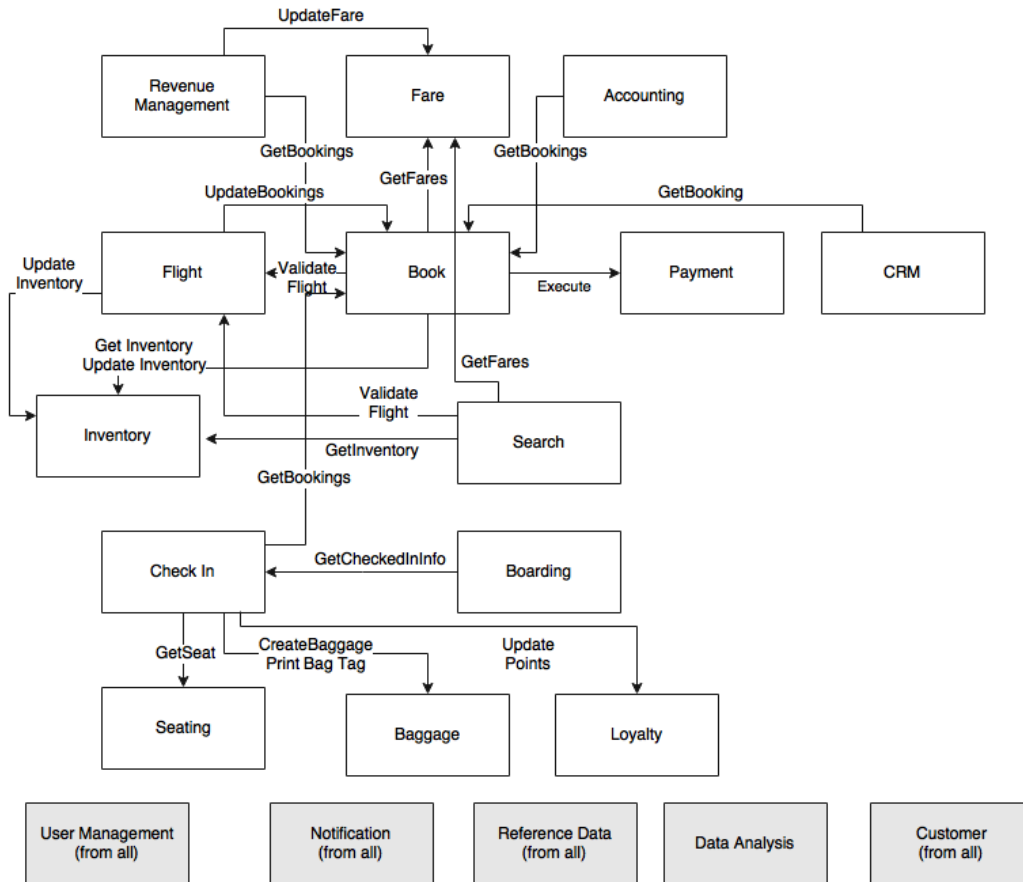


A team of architects, business analysts, developers, release management and support staff is required for this exercise.

One way to produce a dependency graph is to list out all the components of the legacy system and overlay dependencies. This could be done by combining one or more of the approaches listed as follows:

- Analyzing the manual code and regenerating dependencies.
- Using the experience of the development team to regenerate dependencies.
- Using a Maven dependency graph. There are a number of tools we could use to regenerate the dependency graph, such as PomExplorer, PomParser, and so on.
- Using performance engineering tools such as AppDynamics to identify the call stack and roll up dependencies.

Let us assume that we reproduce the functions and their dependencies as shown in the following diagram:



There are many dependencies going back and forth between different modules. The bottom layer shows cross-cutting capabilities that are used across multiple modules. At this point, the modules are more like spaghetti than autonomous units.

The next step is to analyze these dependencies, and come up with a better, simplified dependency map.

Events as opposed to query

Dependencies could be query-based or event-based. Event-based is better for scalable systems. Sometimes, it is possible to convert query-based communications to event-based ones. In many cases, these dependencies exist because either the business organizations are managed like that, or by virtue of the way the old system handled the business scenario.

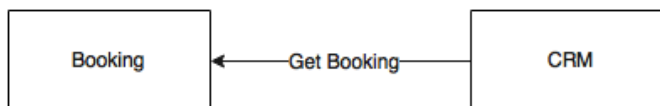
From the previous diagram, we can extract the Revenue Management and the Fares services:



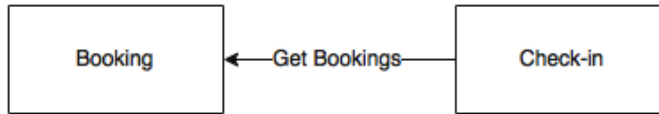
Revenue Management is a module used for calculating optimal fare values, based on the booking demand forecast. In case of a fare change between an origin and a destination, Update Fare on the Fare module is called by Revenue Management to update the respective fares in the Fare module.

An alternate way of thinking is that the Fare module is subscribed to Revenue Management for any changes in fares, and Revenue Management publishes whenever there is a fare change. This reactive programming approach gives an added flexibility by which the Fares and the Revenue Management modules could stay independent, and connect them through a reliable messaging system. This same pattern could be applied in many other scenarios from Check-In to the Loyalty and Boarding modules.

Next, examine the scenario of CRM and Booking:



This scenario is slightly different from the previously explained scenario. The CRM module is used to manage passenger complaints. When CRM receives a complaint, it retrieves the corresponding passenger's Booking data. In reality, the number of complaints are negligibly small when compared to the number of bookings. If we blindly apply the previous pattern where CRM subscribes to all bookings, we will find that it is not cost effective



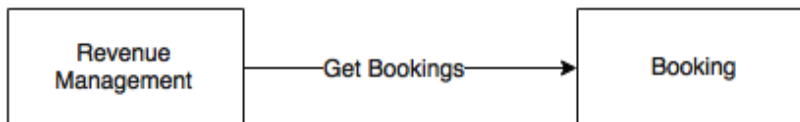
Examine another scenario between the Check-in and Booking modules. Instead of Check-in calling the Get Bookings service on Booking, can Check-in listen to booking events? This is possible, but the challenge here is that a booking can happen 360 days in advance, whereas Check-in generally starts only 24 hours before the flight departure. Duplicating all bookings and booking changes in the Check-in module 360 days in advance would not be a wise decision as Check-in does not require this data until 24 hours before the flight departure

An alternate option is that when check-in opens for a flight (24 hours before departure), Check-in calls a service on the Booking module to get a snapshot of the bookings for a given flight. Once this is done, Check-in could subscribe for booking events specifically for that flight. In this case, a combination of query-based as well as event-based approaches is used. By doing so, we reduce the unnecessary events and storage apart from reducing the number of queries between these two services.

In short, there is no single policy that rules all scenarios. Each scenario requires logical thinking, and then the most appropriate pattern is applied.

Events as opposed to synchronous updates

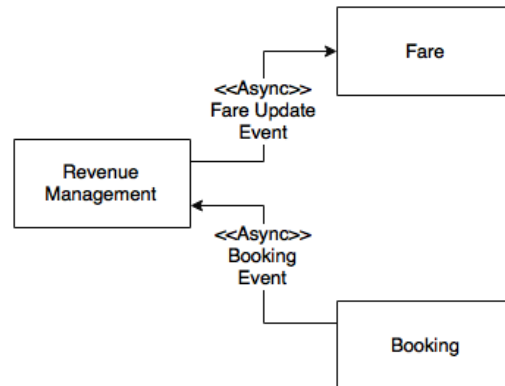
Apart from the query model, a dependency could be an update transaction as well. Consider the scenario between Revenue Management and Booking:



In order to do a forecast and analysis of the current demand, Revenue Management requires all bookings across all flights. The current approach, as depicted in the dependency graph, is that Revenue Management has a schedule job that calls Get Booking on Booking to get all incremental bookings (new and changed) since the last synchronization.

An alternative approach is to send new bookings and the changes in bookings as soon as they take place in the Booking module as an asynchronous push. The same pattern could be applied in many other scenarios such as from Booking to Accounting, from Flight to Inventory, and also from Flight to Booking. In this approach, the source service publishes all state-change events to a topic. All interested parties could subscribe to this event stream and store locally. This approach removes many hard wirings, and keeps the systems loosely coupled.

The dependency is depicted in the next diagram:



In this case depicted in the preceding diagram, we changed both dependencies and converted them to asynchronous events.

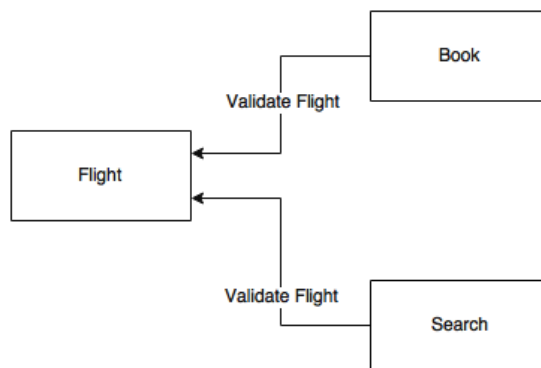
One last case to analyze is the Update Inventory call from the Booking module to the Inventory module:



When a booking is completed, the inventory status is updated by depleting the inventory stored in the Inventory service. For example, when there are 10 economy class seats available, at the end of the booking, we have to reduce it to 9. In the current system, booking and updating inventory are executed within the same transaction boundaries. This is to handle a scenario in which there is only one seat left, and multiple customers are trying to book. In the new design, if we apply the same event-driven pattern, sending the inventory update as an event to Inventory may leave the system in an inconsistent state. This needs further analysis, which we will address later in this chapter.

Challenge requirements

In many cases, the targeted state could be achieved by taking another look at the requirements:



There are two Validate Flight calls, one from Booking and another one from the Search module. The Validate Flight call is to validate the input flight data coming from different channels. The end objective is to avoid incorrect data stored or serviced. When a customer does a flight search, say "BF100", the system validates this flight to see the following things:

- Whether this is a valid flight?
- Whether the flight exists on that particular date?
- Are there any booking restrictions set on this flight?

An alternate way of solving this is to adjust the inventory of the flight based on these given conditions. For example, if there is a restriction on the flight, update the inventory as zero. In this case, the intelligence will remain with Flight, and it keeps updating the inventory. As far as Search and Booking are concerned, both just look up the inventory instead of validating flights for every request. This approach is more efficient as compared to the original approach

Next we will review the Payment use case. Payment is typically a disconnected function due to the nature of security constraints such as PCIDSS-like standards. The most obvious way to capture a payment is to redirect a browser to a payment page hosted in the Payment service. Since card handling applications come under the purview of PCIDSS, it is wise to remove any direct dependencies from the Payment service. Therefore, we can remove the Booking-to-Payment direct dependency, and opt for a UI-level integration.

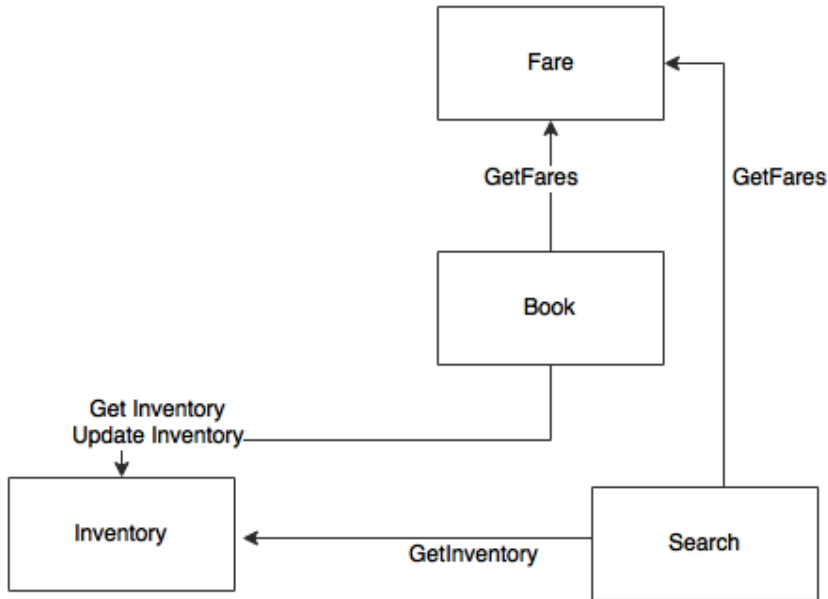
Challenge service boundaries

In this section, we will review some of the service boundaries based on the requirements and dependency graph, considering Check-in and its dependencies to Seating and Baggage.

The Seating function runs a few algorithms based on the current state of the seat allocation in the airplane, and finds out the best way to position the next passenger so that the weight and balance requirements can be met. This is based on a number of predefined business rules. However, other than Check-in, no other module is interested in the Seating function. From a business capability perspective, Seating is just a function of Check-in, not a business capability by itself. Therefore, it is better to embed this logic inside Check-in itself.

The same is applicable to Baggage as well. BrownField has a separate baggage handling system. The Baggage function in the PSS context is to print the baggage tag as well as store the baggage data against the Check-in records. There is no business capability associated with this particular functionality. Therefore, it is ideal to move this function to Check-in itself.

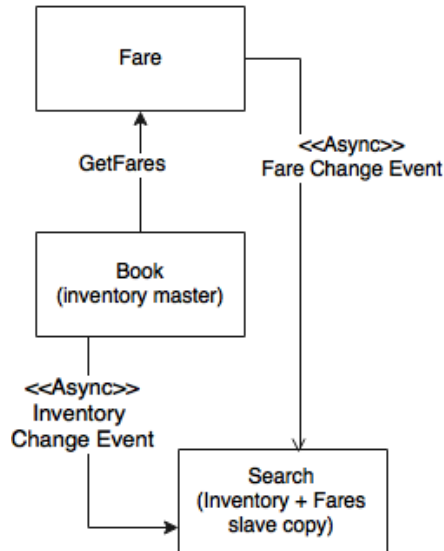
The Book, Search, and Inventory functions, after redesigning, are shown in the following diagram:



Similarly, Inventory and Search are more supporting functions of the Booking module. They are not aligned with any of the business capabilities as such. Similar to the previous judgement, it is ideal to move both the Search and Inventory functions to Booking. Assume, for the time being, that Search, Inventory, and Booking are moved to a single microservice named Reservation.

As per the statistics of BrownField, search transactions are 10 times more frequent than the booking transactions. Moreover, search is not a revenue-generating transaction when compared to booking. Due to these reasons, we need different scalability models for search and booking. Booking should not get impacted if there is a sudden surge of transactions in search. From the business point of view, dropping a search transaction in favor of saving a valid booking transaction is more acceptable.

This is an example of a polyglot requirement, which overrules the business capability alignment. In this case, it makes more sense to have Search as a service separate from the Booking service. Let us assume that we remove Search. Only Inventory and Booking remain under Reservation. Now Search has to hit back to Reservation to perform inventory searches. This could impact the booking transactions:



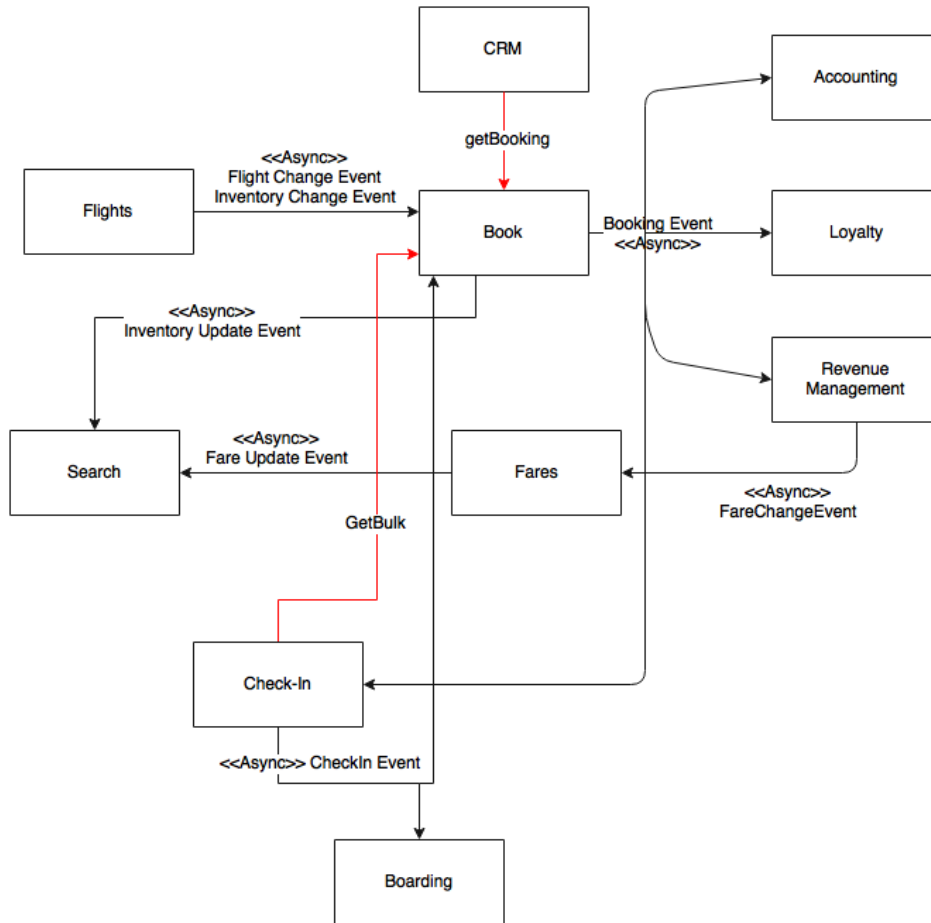
A better approach is to keep Inventory along with the Booking module, and keep a read-only copy of the inventory under Search, while continuously synchronizing the inventory data over a reliable messaging system. Since both Inventory and Booking are collocated, this will also solve the need to have two-phase commits. Since both of them are local, they could work well with local transactions.

Let us now challenge the Fare module design. When a customer searches for a flight between A and B for a given date, we would like to show the flights and fare together. That means that our read-only copy of inventory can also combine both fares as well as inventory. Search will then subscribe to Fare for any fare change events. The intelligence still stays with the Fare service, but it keeps sending fare updates to the cached fare data under Search.

Final dependency graph

There are still a few synchronized calls, which, for the time being, we will keep as they are.

By applying all these changes, the final dependency diagram will look like the following one:



Now we can safely consider each box in the preceding diagram as a microservice. We have nailed down many dependencies, and modeled many of them as asynchronous as well. The overall system is more or less designed in the reactive style. There are still some synchronized calls shown in the diagram with bold lines, such as **Get Bulk** from **Check-In**, **Get Booking** from **CRM**, and **Get Fare** from **Booking**. These synchronous calls are essentially required as per the trade-off analysis.

Prioritizing microservices for migration

We have identified a first-cut version of our microservices-based architecture. As the next step, we will analyze the priorities, and identify the order of migration. This could be done by considering multiple factors explained as follows:

- **Dependency:** One of the parameters for deciding the priority is the dependency graph. From the service dependency graph, services with less dependency or no dependency at all are easy to migrate, whereas complex dependencies are way harder. Services with complex dependencies will also need dependent modules to be migrated along with them.

Accounting, Loyalty, CRM, and Boarding have less dependencies as compared to Booking and Check-in. Modules with high dependencies will also have higher risks in their migration.

- **Transaction volume:** Another parameter that can be applied is analyzing the transaction volumes. Migrating services with the highest transaction volumes will relieve the load on the existing system. This will have more value from an IT support and maintenance perspective. However, the downside of this approach is the higher risk factor.

As stated earlier, Search requests are ten times higher in volume as compared to Booking requests. Requests for Check-in are the third-highest in volume transaction after Search and Booking.

- **Resource utilization:** Resource utilization is measured based on the current utilizations such as CPU, memory, connection pools, thread pools, and so on. Migrating resource intensive services out of the legacy system provides relief to other services. This helps the remaining modules to function better.

Flight, Revenue Management, and Accounting are resource-intensive services, as they involve data-intensive transactions such as forecasting, billing, flight schedule changes, and so on.

- **Complexity:** Complexity is perhaps measured in terms of the business logic associated with a service such as function points, lines of code, number of tables, number of services, and others. Less complex modules are easy to migrate as compared to the more complex ones.

Booking is extremely complex as compared to the Boarding, Search, and Check-in services.

- **Business criticality:** The business criticality could be either based on revenue or customer experience. Highly critical modules deliver higher business value.

Booking is the most revenue-generating service from the business stand point, whereas Check-in is business critical as it could lead to flight departure delays, which could lead to revenue loss as well as customer dissatisfaction.

- **Velocity of changes:** Velocity of change indicates the number of change requests targeting a function in a short time frame. This translates to speed and agility of delivery. Services with high velocity of change requests are better candidates for migration as compared to stable modules.

Statistics show that Search, Booking, and Fares go through frequent changes, whereas Check-in is the most stable function.

- **Innovation:** Services that are part of a disruptive innovative process need to get priority over back office functions that are based on more established business processes. Innovations in legacy systems are harder to achieve as compared to applying innovations in the microservices world.

Most of the innovations are around Search, Booking, Fares, Revenue Management, and Check-in as compared to back office Accounting.

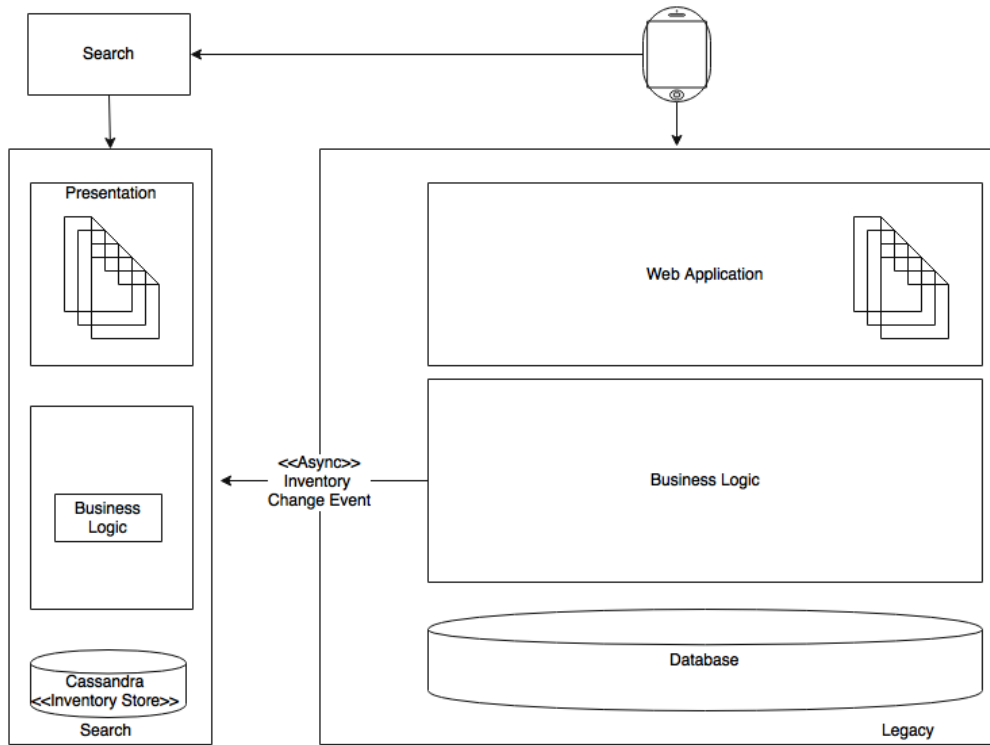
Based on BrownField's analysis, Search has the highest priority, as it requires innovation, has high velocity of changes, is less business critical, and gives better relief for both business and IT. The Search service has minimal dependency with no requirements to synchronize data back to the legacy system.

Data synchronization during migration

During the transition phase, the legacy system and the new microservices will run in parallel. Therefore, it is important to keep the data synchronized between the two systems.

The simplest option is to synchronize the data between the two systems at the database level by using any data synchronization tool. This approach works well when both the old and the new systems are built on the same data store technologies. The complexity will be higher if the data store technologies are different. The second problem with this approach is that we allow a backdoor entry, hence exposing the microservices' internal data store outside. This is against the principle of microservices.

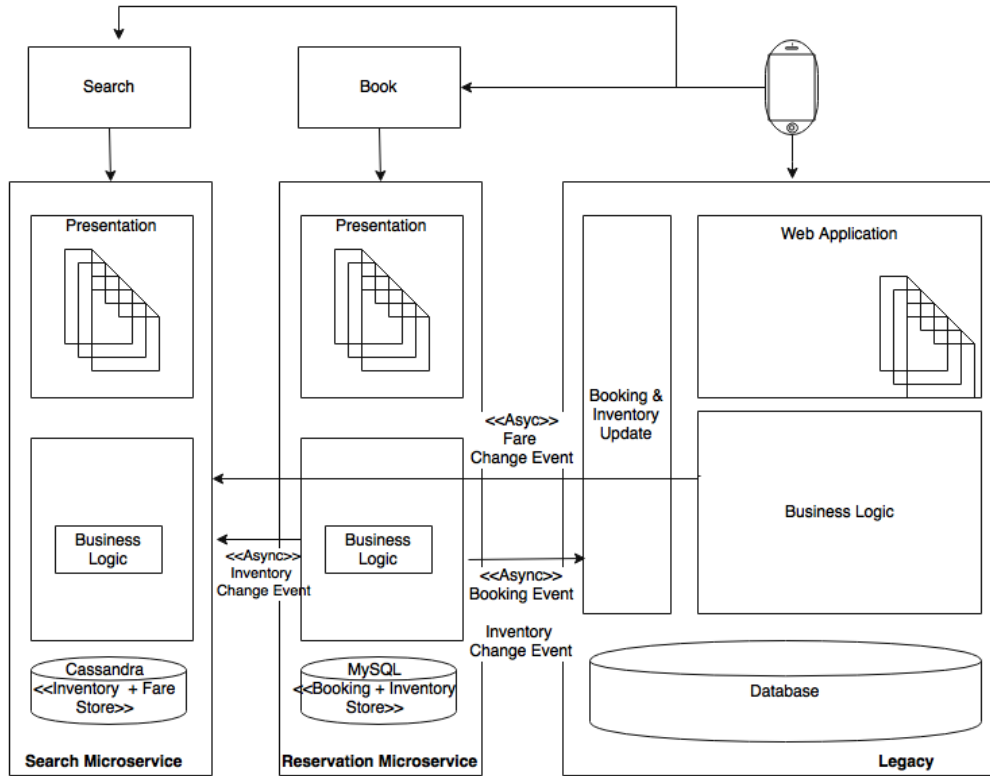
Let us take this on a case-by-case basis before we can conclude with a generic solution. The following diagram shows the data migration and synchronization aspect once Search is taken out:



Let us assume that we use a NoSQL database for keeping inventory and fares under the Search service. In this particular case, all we need is the legacy system to supply data to the new service using asynchronous events. We will have to make some changes in the existing system to send the fare changes or any inventory changes as events. The Search service then accepts these events, and stores them locally into the local NoSQL store.

This is a bit more tedious in the case of the complex Booking service.

In this case, the new Booking microservice sends the inventory change events to the Search service. In addition to this, the legacy application also has to send the fare change events to Search. Booking will then store the new Booking service in its My SQL data store.

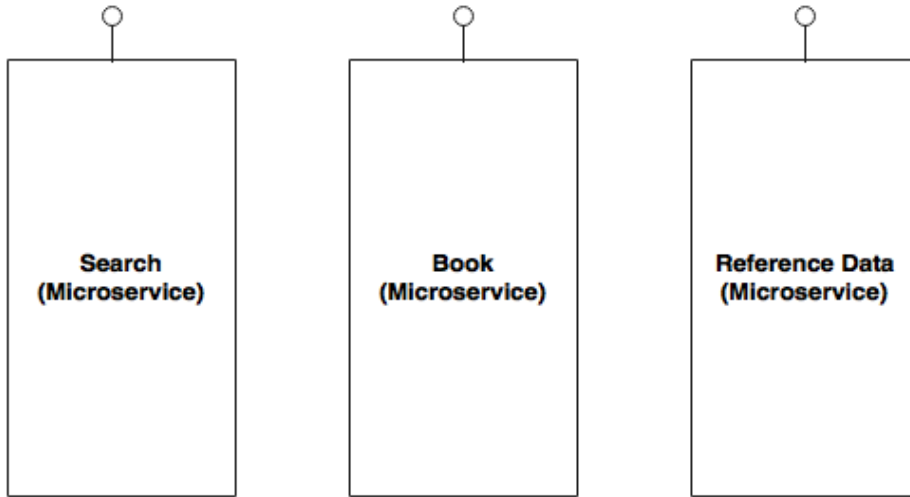


The most complex piece, the Booking service, has to send the booking events and the inventory events back to the legacy system. This is to ensure that the functions in the legacy system continue to work as before. The simplest approach is to write an update component which accepts the events and updates the old booking records table so that there are no changes required in the other legacy modules. We will continue this until none of the legacy components are referring the booking and inventory data. This will help us minimize changes in the legacy system, and therefore, reduce the risk of failures.

In short, a single approach may not be sufficient. A multi-pronged approach based on different patterns is required.

Managing reference data

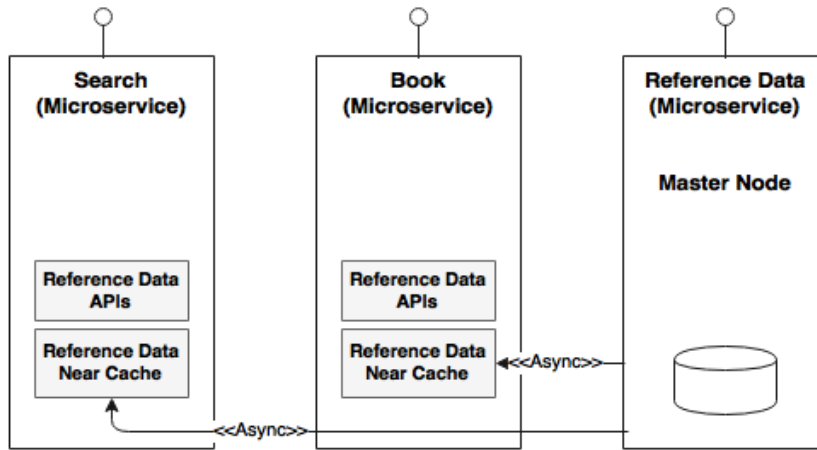
One of the biggest challenges in migrating monolithic applications to microservices is managing reference data. A simple approach is to build the reference data as another microservice itself as shown in the following diagram:



In this case, whoever needs reference data should access it through the microservice endpoints. This is a well-structured approach, but could lead to performance issues as encountered in the original legacy system.

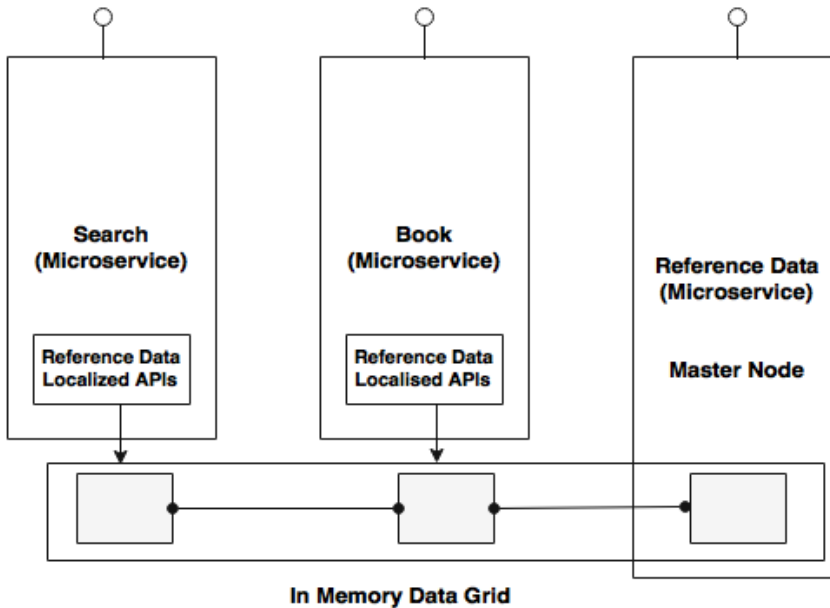
An alternate approach is to have reference data as a microservice service for all the admin and CRUD functions. A near cache will then be created under each service to incrementally cache data from the master services. A thin reference data access proxy library will be embedded in each of these services. The reference data access proxy abstracts whether the data is coming from cache or from a remote service.

This is depicted in the next diagram. The master node in the given diagram is the actual reference data microservice:



The challenge is to synchronize the data between the master and the slave. A subscription mechanism is required for those data caches that change frequently.

A better approach is to replace the local cache with an in-memory data grid, as shown in the following diagram:

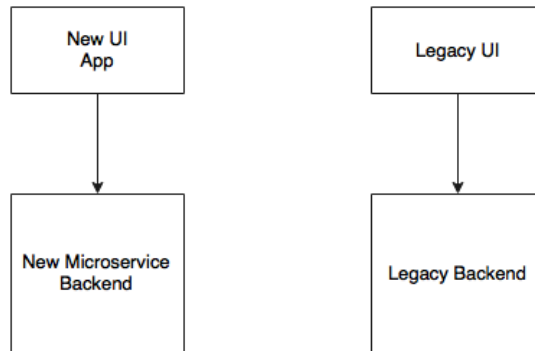


The reference data microservice will write to the data grid, whereas the proxy libraries embedded in other services will have read-only APIs. This eliminates the requirement to have subscription of data, and is much more efficient and consistent

User interfaces and web applications

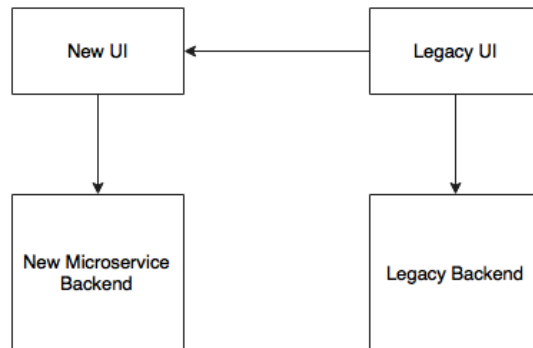
During the transition phase, we have to keep both the old and new user interfaces together. There are three general approaches usually taken in this scenario.

The first approach is to have the old and new user interfaces as separate user applications with no link between them, as depicted in the following diagram:



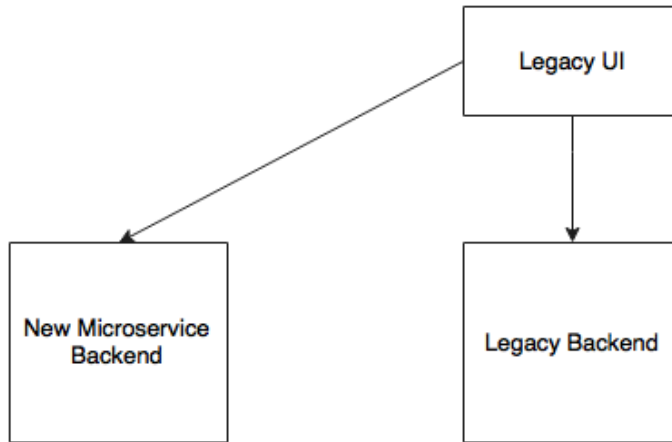
A user signs in to the new application as well as into the old application, much like two different applications, with no **single sign-on (SSO)** between them. This approach is simple, and there is no overhead. In most of the cases, this may not be acceptable to the business unless it is targeted at two different user communities.

The second approach is to use the legacy user interface as the primary application, and then transfer page controls to the new user interfaces when the user requests pages of the new application:



In this case, since the old and the new applications are web-based applications running in a web browser window, users will get a seamless experience. SSO has to be implemented between the old and the new user interfaces.

The third approach is to integrate the existing legacy user interface directly to the new microservices backend, as shown in the next diagram:



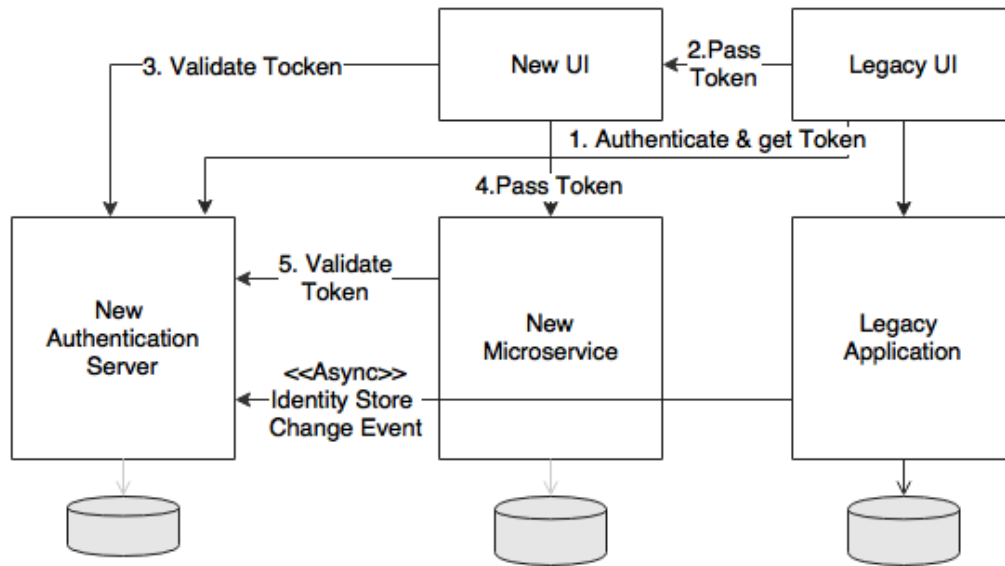
In this case, the new microservices are built as headless applications with no presentation layer. This could be challenging, as it may require many changes in the old user interface such as introducing service calls, data model conversions, and so on.

Another issue in the last two cases is how to handle the authentication of resources and services.

Session handling and security

Assume that the new services are written based on Spring Security with a token-based authorization strategy, whereas the old application uses a custom-built authentication with its local identity store.

The following diagram shows how to integrate between the old and the new services:



The simplest approach, as shown in the preceding diagram, is to build a new identity store with an authentication service as a new microservice using Spring Security. This will be used for all our future resource and service protections, for all microservices.

The existing user interface application authenticates itself against the new authentication service, and secures a token. This token will be passed to the new user interface or new microservice. In both cases, the user interface or microservice will make a call to the authentication service to validate the given token. If the token is valid, then the UI or microservice accepts the call.

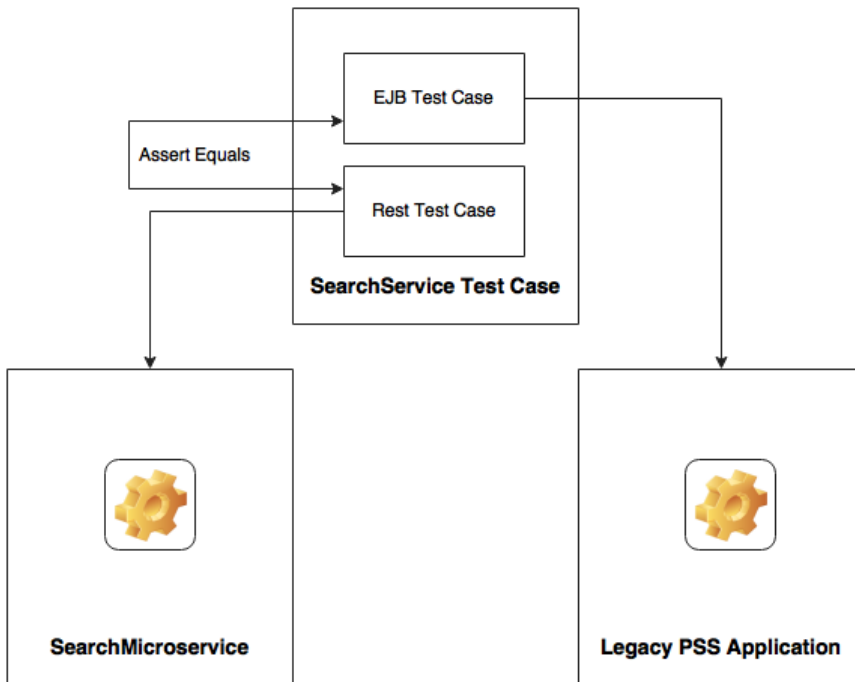
The catch here is that the legacy identity store has to be synchronized with the new one.

Test strategy

One important question to answer from a testing point of view is how can we ensure that all functions work in the same way as before the migration?

Integration test cases should be written for the services that are getting migrated before the migration or refactoring. This ensures that once migrated, we get the same expected result, and the behavior of the system remains the same. An automated regression test pack has to be in place, and has to be executed every time we make a change in the new or old system.

In the following diagram, for each service we need one test against the EJB endpoint, and another one against the microservices endpoint:



Building ecosystem capabilities

Before we embark on actual migration, we have to build all of the microservice's capabilities mentioned under the capability model, as documented in *Chapter 3, Applying Microservices Concepts*. These are the prerequisites for developing microservices-based systems.

In addition to these capabilities, certain application functions are also required to be built upfront such as reference data, security and SSO, and Customer and Notification. A data warehouse or a data lake is also required as a prerequisite. An effective approach is to build these capabilities in an incremental fashion, delaying development until it is really required.

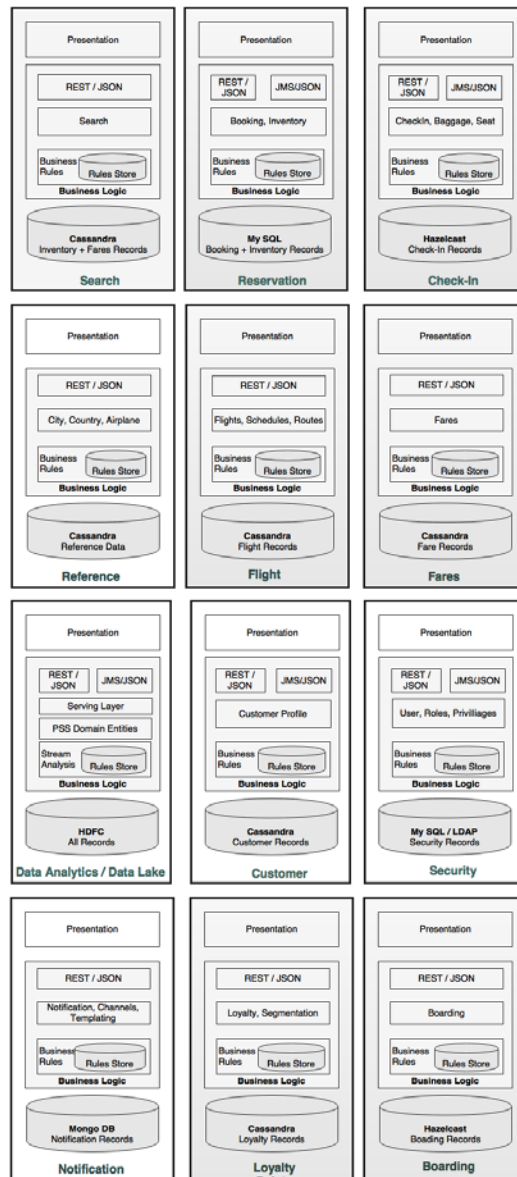
Migrate modules only if required

In the previous chapters, we have examined approaches and steps for transforming from a monolithic application to microservices. It is important to understand that it is not necessary to migrate all modules to the new microservices architecture, unless it is really required. A major reason is that these migrations incur cost.

We will review a few such scenarios here. BrownField has already taken a decision to use an external revenue management system in place of the PSS revenue management function. BrownField is also in the process of centralizing their accounting functions, and therefore, need not migrate the accounting function from the legacy system. Migration of CRM does not add much value at this point to the business. Therefore, it is decided to keep the CRM in the legacy system itself. The business has plans to move to a SaaS-based CRM solution as part of their cloud strategy. Also note that stalling the migration halfway through could seriously impact the complexity of the system.

Target architecture

The architecture blueprint shown in the following diagram consolidates earlier discussions into an architectural view. Each block in the diagram represents a microservice. The shaded boxes are core microservices, and the others are supporting microservices. The diagram also shows the internal capabilities of each microservice. User management is moved under security in the target architecture:

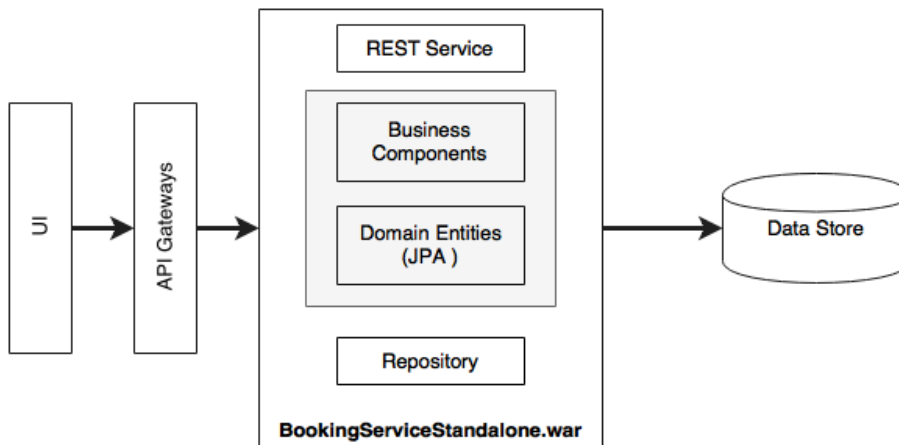


Each service has its own architecture, typically consisting of a presentation layer, one or more service endpoints, business logic, business rules, and database. As we can see, we use different selections of databases that are more suitable for each microservice. Each one is autonomous with minimal orchestration between the services. Most of the services interact with each other using the service endpoints.

Internal layering of microservices

In this section, we will further explore the internal structure of microservices. There is no standard to be followed for the internal architecture of a microservice. The rule of thumb is to abstract realizations behind simple service endpoints.

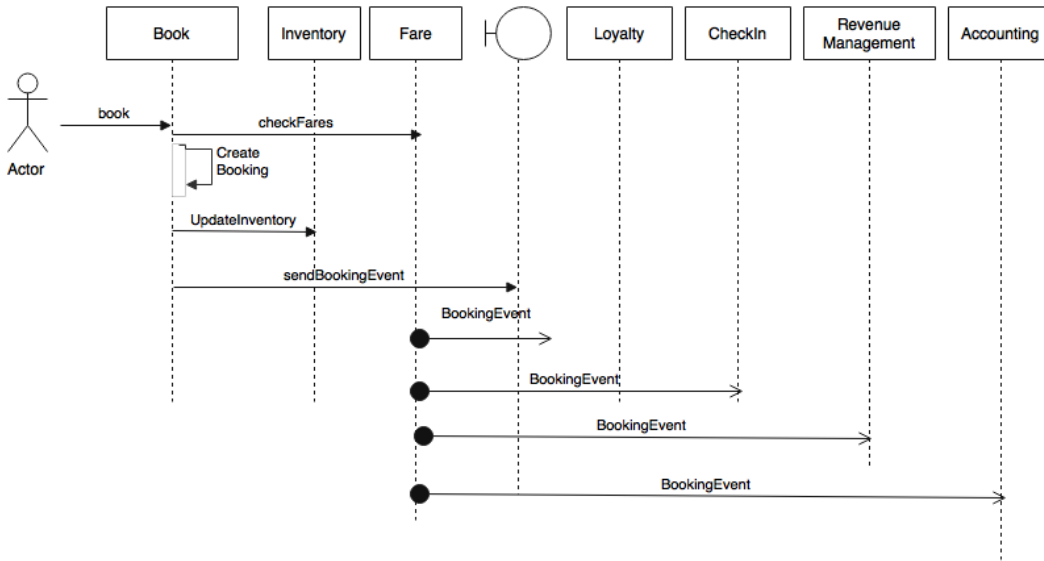
A typical structure would look like the one shown in the following diagram:



The UI accesses REST services through a service gateway. The API gateway may be one per microservice or one for many microservices – it depends on what we want to do with the API gateway. There could be one or more rest endpoints exposed by microservices. These endpoints, in turn, connect to one of the business components within the service. Business components then execute all the business functions with the help of domain entities. A repository component is used for interacting with the backend data store.

Orchestrating microservices

The logic of the booking orchestration and the execution of rules sits within the Booking service. The brain is still inside the Booking service in the form of one or more booking business components. Internally, business components orchestrate private APIs exposed by other business components or even external services:



As shown in the preceding diagram, the booking service internally calls to update the inventory of its own component other than calling the Fare service.

Is there any orchestration engine required for this activity? It depends on the requirements. In complex scenarios, we may have to do a number of things in parallel. For example, creating a booking internally applies a number of booking rules, it validates the fare, and it validates the inventory before creating a booking. We may want to execute them in parallel. In such cases, we may use Java concurrency APIs or reactive Java libraries.

In extremely complex situations, we may opt for an integration framework such as Spring Integration or Apache Camel in embedded mode.

Integration with other systems

In the microservices world, we use an API gateway or a reliable message bus for integrating with other non-microservices.

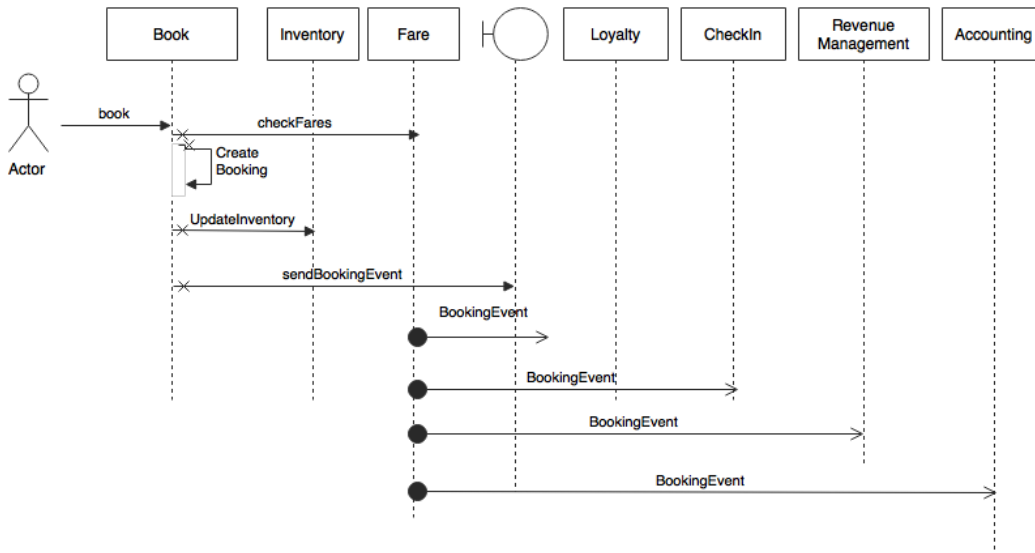
Let us assume that there is another system in BrownField that needs booking data. Unfortunately, the system is not capable of subscribing to the booking events that the Booking microservice publishes. In such cases, an **Enterprise Application integration (EAI)** solution could be employed, which listens to our booking events, and then uses a native adaptor to update the database.

Managing shared libraries

Certain business logic is used in more than one microservice. Search and Reservation, in this case, use inventory rules. In such cases, these shared libraries will be duplicated in both the microservices.

Handling exceptions

Examine the booking scenario to understand the different exception handling approaches. In the following service sequence diagram, there are three lines marked with a cross mark. These are the potential areas where exceptions could occur:

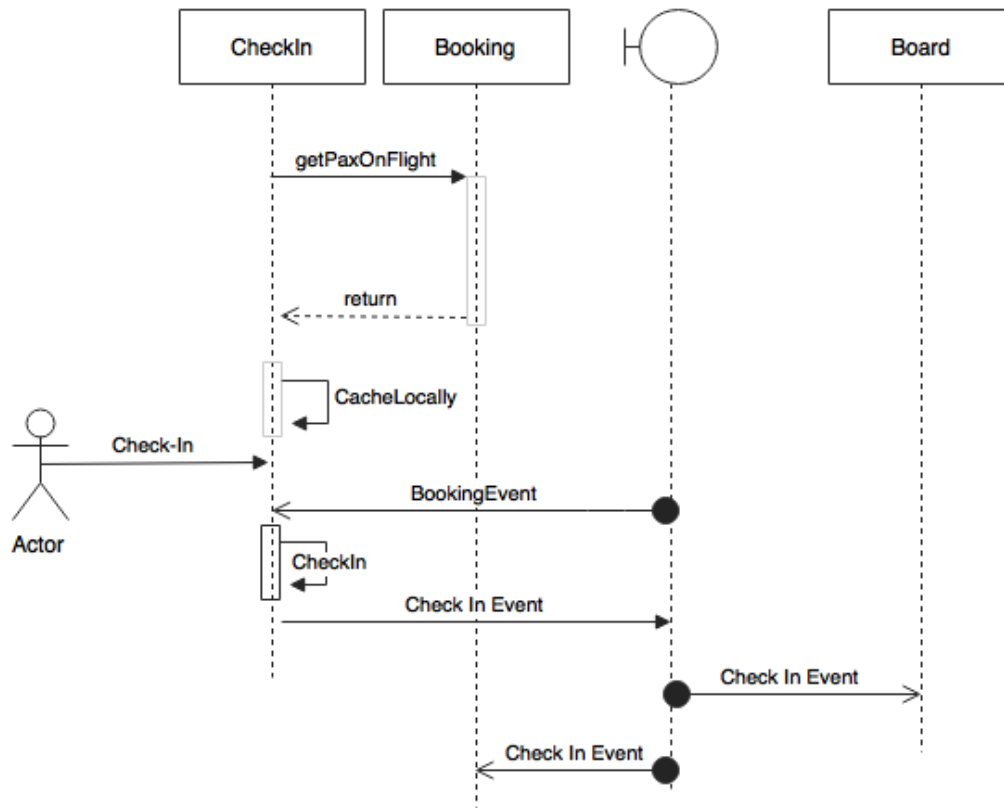


There is a synchronous communication between Booking and Fare. What if the Fare service is not available? If the Fare service is not available, throwing an error back to the user may cause revenue loss. An alternate thought is to trust the fare which comes as part of the incoming request. When we serve search, the search results will have the fare as well. When the user selects a flight and submits, the request will have the selected fare. In case the Fare service is not available, we trust the incoming request, and accept the Booking. We will use a circuit breaker and a fallback service which simply creates the booking with a special status, and queues the booking for manual action or a system retry.

What if creating the booking fails? If creating a booking fails unexpectedly, a better option is to throw a message back to the user. We could try alternative options, but that could increase the overall complexity of the system. The same is applicable for inventory updates.

In the case of creating a booking and updating the inventory, we avoid a situation where a booking is created, and an inventory update somehow fails. As the inventory is critical, it is better to have both, create booking and update inventory, to be in a local transaction. This is possible as both components are under the same subsystem.

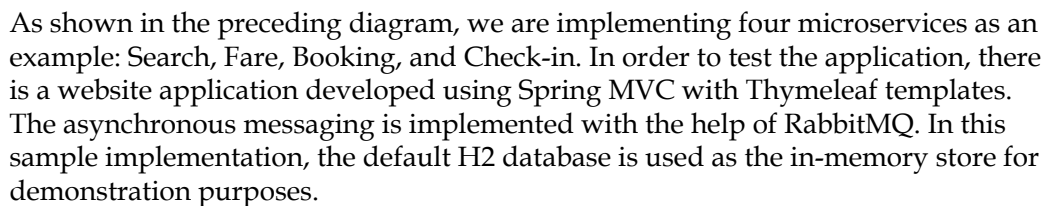
If we consider the Check-in scenario, Check-in sends an event to Boarding and Booking as shown in the next diagram:



Consider a scenario where the Check-in services fail immediately after the Check-in Complete event is sent out. The other consumers processed this event, but the actual check-in is rolled back. This is because we are not using a two-phase commit. In this case, we need a mechanism for reverting that event. This could be done by catching the exception, and sending another Check-in Cancelled event.

In this case, note that to minimize the use of compensating transactions, sending the Check-in event is moved towards the end of the Check-in transaction. This reduces the chance of failure after sending out the event.

The next diagram represents the implementation view of the BrownField PSS microservices system:



- [194]

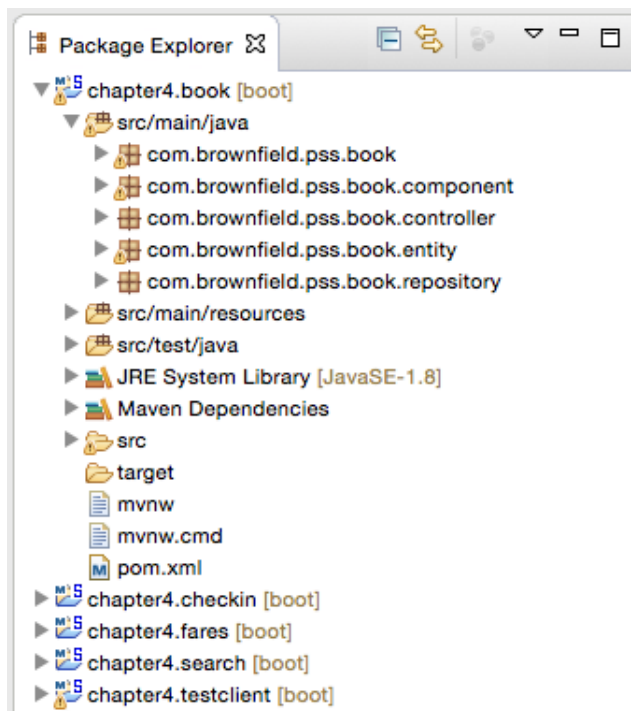
Implementation projects

The basic implementation of the BrownField Airline's PSS microservices system has five core projects as summarized in the following table. The table also shows the port range used for these projects to ensure consistency throughout the book:

Microservice	Projects	Port Range
Book microservice	chapter4.book	8060-8069
Check-in microservice	chapter4.checkin	8070-8079
Fare microservice	chapter4.fares	8080-8089
Search microservice	chapter4.search	8090-8099
Website	chapter4.website	8001

The website is the UI application for testing the PSS microservices.

All microservice projects in this example follow the same pattern for package structure as shown in the following screenshot:



The different packages and their purposes are explained as follows:

- The root folder (`com.brownfield.pss.book`) contains the default Spring Boot application.
- The `component` package hosts all the service components where the business logic is implemented.
- The `controller` package hosts the REST endpoints and the messaging endpoints. Controller classes internally utilize the component classes for execution.
- The `entity` package contains the JPA entity classes for mapping to the database tables.
- Repository classes are packaged inside the `repository` package, and are based on Spring Data JPA.

Running and testing the project

Follow the steps listed next to build and test the microservices developed in this chapter:

1. Build each of the projects using Maven. Ensure that the `test` flag is switched off. The test programs assume other dependent services are up and running. It fails if the dependent services are not available. In our example, Booking and Fare have direct dependencies. We will learn how to circumvent this dependency in *Chapter 7, Logging and Monitoring Microservices*:

```
mvn -Dmaven.test.skip=true install
```

2. Run the RabbitMQ server:

```
rabbitmq_server-3.5.6/sbin$ ./rabbitmq-server
```

3. Run the following commands in separate terminal windows:

```
java -jar target/fares-1.0.jar
```

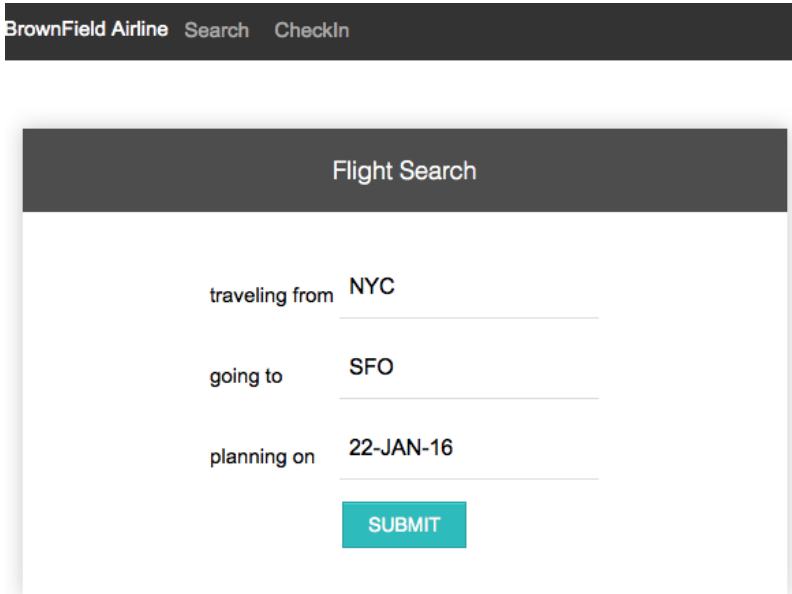
```
java -jar target/search-1.0.jar
```

```
java -jar target/checkin-1.0.jar
```

```
java -jar target/book-1.0.jar
```

```
java -jar target/website-1.0.jar
```

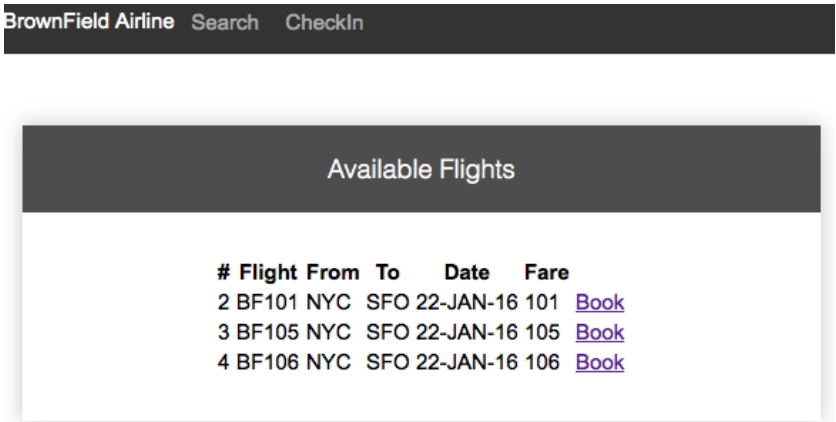
4. The website project has a `CommandLineRunner`, which executes all the test cases at startup. Once all the services are successfully started, open `http://localhost:8001` in a browser.
5. The browser asks for basic security credentials. Use `guest` or `guest123` as the credentials. This example only shows the website security with a basic authentication mechanism. As explained in *Chapter 2, Building Microservices with Spring Boot*, service-level security can be achieved using `OAuth2`.
6. Entering the correct security credentials displays the following screen. This is the home screen of our `BrownField PSS` application:



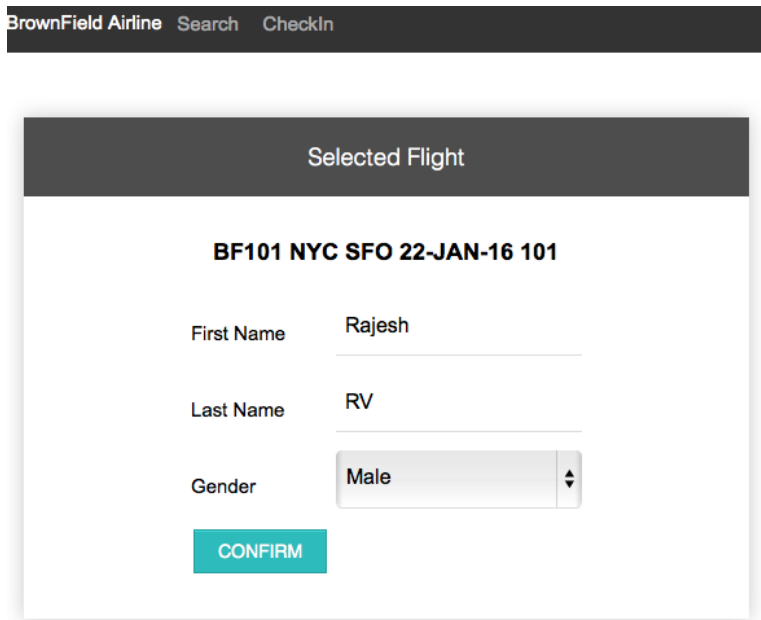
The screenshot shows the user interface of the BrownField Airline application. At the top, there is a dark header bar with the text "BrownField Airline" in white, followed by "Search" and "CheckIn" in a lighter color. Below this header is a white box with a dark header titled "Flight Search". Inside this box, there are three input fields with labels "traveling from", "going to", and "planning on". The values entered in these fields are "NYC", "SFO", and "22-JAN-16" respectively. Below the input fields is a teal button labeled "SUBMIT".

7. The **SUBMIT** button invokes the `Search` microservice to fetch the available flight that meet the conditions mentioned on the screen. A few flights are pre-populated at the startup of the `Search` microservice. Edit the `Search` microservice code to feed in additional flights, if required

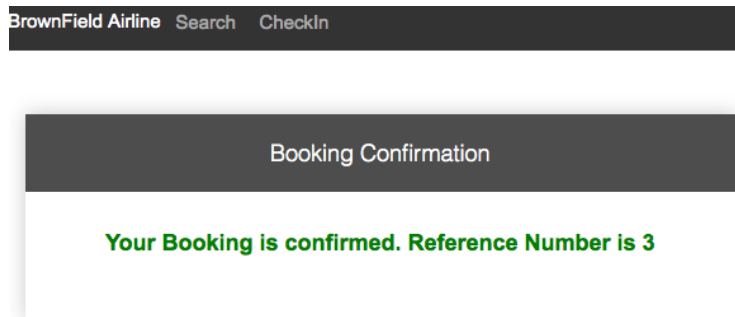
8. The output screen with a list of flights is shown in the next screenshot. The **Book** link will take us to the booking screen for the selected flight



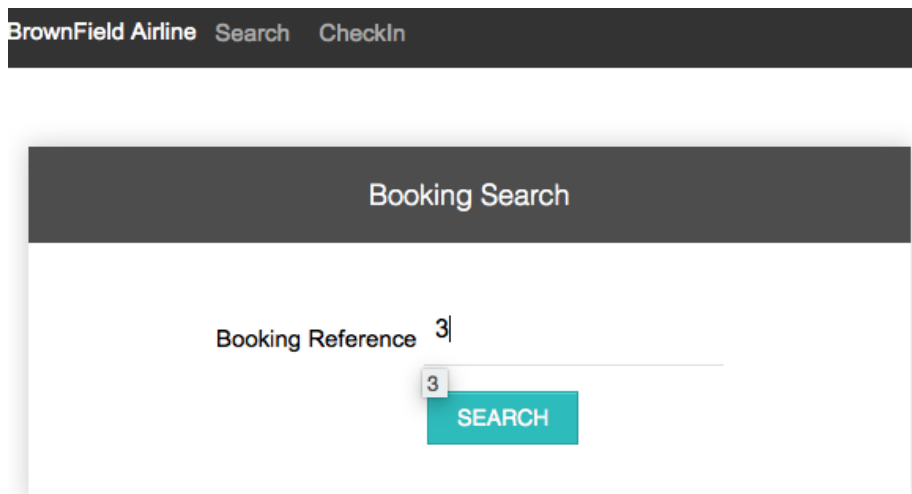
9. The following screenshot shows the booking screen. The user can enter the passenger details, and create a booking by clicking on the **CONFIRM** button. This invokes the Booking microservice, and internally, the Fare service as well. It also sends a message back to the Search microservice:



10. If booking is successful, the next confirmation screen is displayed with a booking reference number:



11. Let us test the Check-in microservice. This can be done by clicking on **CheckIn** in the menu at the top of the screen. Use the booking reference number obtained in the previous step to test Check-in. This is shown in the following screenshot:



12. Clicking on the **SEARCH** button in the previous screen invokes the Booking microservice, and retrieves the booking information. Click on the **CheckIn** link to perform the check-in. This invokes the Check-in microservice:

The screenshot shows the 'Booking Search' interface. At the top is a dark navigation bar with 'BrownField Airline', 'Search', and 'CheckIn' links. Below this is a white box with a dark header 'Booking Search'. Inside the box, there is a label 'Booking Reference' followed by the number '3'. Below the label is a teal 'SEARCH' button. At the bottom of the box, the text 'BF101 NYC SFO 22-JAN-16 Rajesh RV' is displayed, followed by a blue 'CheckIn' link.

13. If check-in is successful, it displays the confirmation message, as shown in the next screenshot, with a confirmation number. This is done by calling the Check-in service internally. The Check-in service sends a message to Booking to update the check-in status:

The screenshot shows the 'Check In Confirmation' interface. At the top is a dark navigation bar with 'BrownField Airline', 'Search', and 'CheckIn' links. Below this is a white box with a dark header 'Check In Confirmation'. Inside the box, the text 'Checked In, Seat Number is 28c , checkin id is 3' is displayed in green.

Summary

In this chapter, we implemented and tested the BrownField PSS microservice with basic Spring Boot capabilities. We learned how to approach a real use case with a microservices architecture.

We examined the various stages of a real-world evolution towards microservices from a monolithic application. We also evaluated the pros and cons of multiple approaches, and the obstacles encountered when migrating a monolithic application. Finally, we explained the end-to-end microservices design for the use case that we examined. Design and implementation of a fully-fledged microservice implementation was also validated.

In the next chapter, we will see how the Spring Cloud project helps us to transform the developed BrownField PSS microservices to an Internet-scale deployment.

5

Scaling Microservices with Spring Cloud

In order to manage Internet-scale microservices, one requires more capabilities than what are offered by the Spring Boot framework. The Spring Cloud project has a suite of purpose-built components to achieve these additional capabilities effortlessly.

This chapter will provide a deep insight into the various components of the Spring Cloud project such as Eureka, Zuul, Ribbon, and Spring Config by positioning them against the microservices capability model discussed in *Chapter 3, Applying Microservices Concepts*. It will demonstrate how the Spring Cloud components help to scale the BrownField Airline's PSS microservices system, developed in the previous chapter.

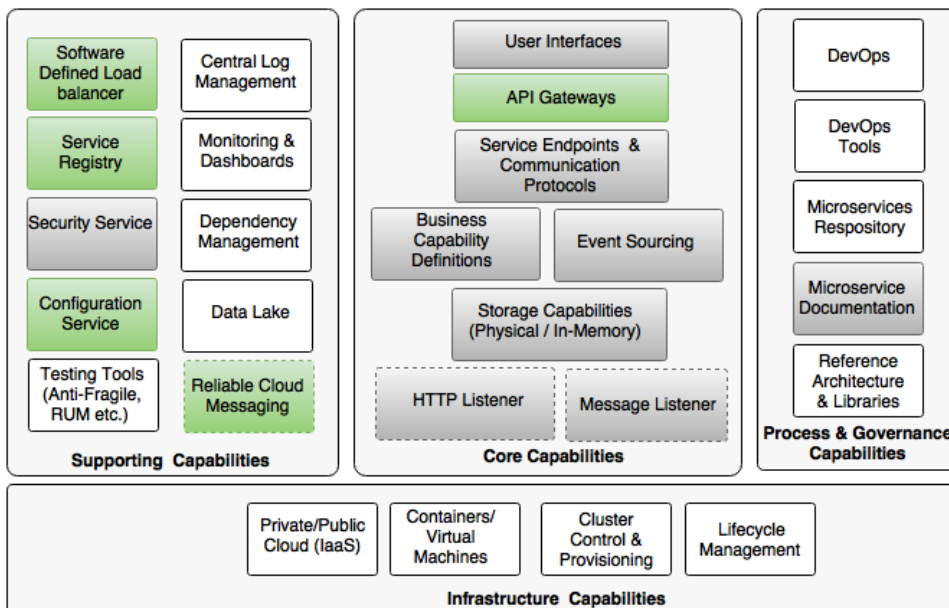
By the end of this chapter, you will learn about the following:

- The Spring Config server for externalizing configuration
- The Eureka server for service registration and discovery
- The relevance of Zuul as a service proxy and gateway
- The implementation of automatic microservice registration and service discovery
- Spring Cloud messaging for asynchronous microservice composition

Reviewing microservices capabilities

The examples in this chapter explore the following microservices capabilities from the microservices capability model discussed in *Chapter 3, Applying Microservices Concepts*:

- **Software Defined Load Balancer**
- **Service Registry**
- **Configuration Service**
- **Reliable Cloud Messaging**
- **API Gateways**



Reviewing BrownField's PSS implementation

In *Chapter 4, Microservices Evolution – A Case Study*, we designed and developed a microservice-based PSS system for BrownField Airlines using the Spring framework and Spring Boot. The implementation is satisfactory from the development point of view, and it serves the purpose for low volume transactions. However, this is not good enough for deploying large, enterprise-scale deployments with hundreds or even thousands of microservices.

In *Chapter 4, Microservices Evolution – A Case Study*, we developed four microservices: Search, Booking, Fares, and Check-in. We also developed a website to test the microservices.

We have accomplished the following items in our microservice implementation so far:

- Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities
- Each microservice implements certain business functions using the Spring framework.
- Each microservice stores its own persistent data using H2, an in-memory database
- Microservices are built with Spring Boot, which has an embedded Tomcat server as the HTTP listener
- RabbitMQ is used as an external messaging service. Search, Booking, and Check-in interact with each other through asynchronous messaging
- Swagger is integrated with all microservices for documenting the REST APIs.
- An OAuth2-based security mechanism is developed to protect the microservices

What is Spring Cloud?

The Spring Cloud project is an umbrella project from the Spring team that implements a set of common patterns required by distributed systems, as a set of easy-to-use Java Spring libraries. Despite its name, Spring Cloud by itself is not a cloud solution. Rather, it provides a number of capabilities that are essential when developing applications targeting cloud deployments that adhere to the Twelve-Factor application principles. By using Spring Cloud, developers just need to focus on building business capabilities using Spring Boot, and leverage the distributed, fault-tolerant, and self-healing capabilities available out of the box from Spring Cloud.

The Spring Cloud solutions are agnostic to the deployment environment, and can be developed and deployed in a desktop PC or in an elastic cloud. The cloud-ready solutions that are developed using Spring Cloud are also agnostic and portable across many cloud providers such as Cloud Foundry, AWS, Heroku, and so on. When not using Spring Cloud, developers will end up using services natively provided by the cloud vendors, resulting in deep coupling with the PaaS providers. An alternate option for developers is to write quite a lot of boilerplate code to build these services. Spring Cloud also provides simple, easy-to-use Spring-friendly APIs, which abstract the cloud provider's service APIs such as those APIs coming with AWS Notification Service

Built on Spring's "convention over configuration" approach, Spring Cloud defaults all configurations, and helps the developers get off to a quick start. Spring Cloud also hides the complexities, and provides simple declarative configurations to build systems. The smaller footprints of the Spring Cloud components make it developer friendly, and also make it easy to develop cloud-native applications.

Spring Cloud offers many choices of solutions for developers based on their requirements. For example, the service registry can be implemented using popular options such as Eureka, ZooKeeper, or Consul. The components of Spring Cloud are fairly decoupled, hence, developers get the flexibility to pick and choose what is required.



What is the difference between Spring Cloud and Cloud Foundry?

Spring Cloud is a developer kit for developing Internet-scale Spring Boot applications, whereas Cloud Foundry is an open-source Platform as a Service for building, deploying, and scaling applications.

Spring Cloud releases

The Spring Cloud project is an overarching Spring project that includes a combination of different components. The versions of these components are defined in the `spring-cloud-starter-parent` BOM.

In this book, we are relying on the `Brixton.RELEASE` version of the Spring Cloud:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-dependencies</artifactId>
  <version>Brixton.RELEASE</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

The `spring-cloud-starter-parent` defines different versions of its subcomponents as follows:

```
<spring-cloud-aws.version>1.1.0.RELEASE</spring-cloud-aws.version>
<spring-cloud-bus.version>1.1.0.RELEASE</spring-cloud-bus.version>
<spring-cloud-cloudfoundry.version>1.0.0.RELEASE</spring-cloud-cloudfoundry.version>
<spring-cloud-commons.version>1.1.0.RELEASE</spring-cloud-commons.version>
<spring-cloud-config.version>1.1.0.RELEASE</spring-cloud-config.version>
<spring-cloud-netflix.version>1.1.0.RELEASE</spring-cloud-netflix.version>
```

```

<spring-cloud-security.version>1.1.0.RELEASE</spring-cloud-security.
version>
<spring-cloud-cluster.version>1.0.0.RELEASE</spring-cloud-cluster.
version>
<spring-cloud-consul.version>1.0.0.RELEASE</spring-cloud-consul.
version>
<spring-cloud-sleuth.version>1.0.0.RELEASE</spring-cloud-sleuth.
version>
<spring-cloud-stream.version>1.0.0.RELEASE</spring-cloud-stream.
version>
<spring-cloud-zookeeper.version>1.0.0.RELEASE </spring-cloud-
zookeeper.version>

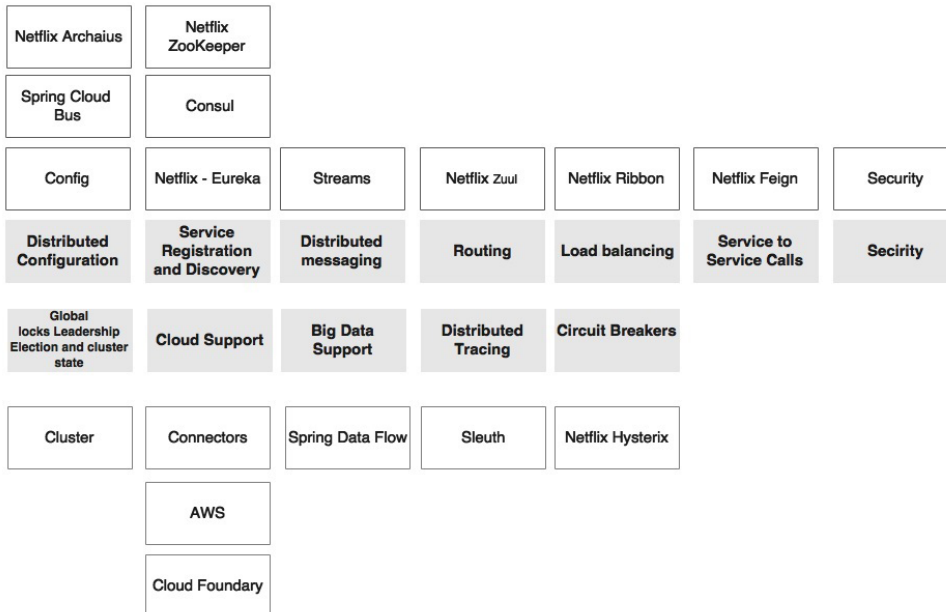
```



The names of the Spring Cloud releases are in an alphabetic sequence, starting with A, following the names of the London Tube stations. **Angel** was the first release, and **Brixton** is the second release.

Components of Spring Cloud

Each Spring Cloud component specifically addresses certain distributed system capabilities. The grayed-out boxes at the bottom of the following diagram show the capabilities, and the boxes placed on top of these capabilities showcase the Spring Cloud subprojects addressing these capabilities:



The Spring Cloud capabilities are explained as follows:

- **Distributed configuration:** Configuration properties are hard to manage when there are many microservice instances running under different profiles such as development, test, production, and so on. It is, therefore, important to manage them centrally, in a controlled way. The distributed configuration management module is to externalize and centralize microservice configuration parameters. Spring Cloud Config is an externalized configuration server with Git or SVN as the backing repository. Spring Cloud Bus provides support for propagating configuration changes to multiple subscribers, generally a microservice instance. Alternately, ZooKeeper or HashiCorp's Consul can also be used for distributed configuration management.
- **Routing:** Routing is an API gateway component, primarily used similar to a reverse proxy that forwards requests from consumers to service providers. The gateway component can also perform software-based routing and filtering. Zuul is a lightweight API gateway solution that offers fine-grained controls to developers for traffic shaping and request/response transformations.
- **Load balancing:** The load balancer capability requires a software-defined load balancer module which can route requests to available servers using a variety of load balancing algorithms. Ribbon is a Spring Cloud subproject which supports this capability. Ribbon can work as a standalone component, or integrate and work seamlessly with Zuul for traffic routing.
- **Service registration and discovery:** The service registration and discovery module enables services to programmatically register with a repository when a service is available and ready to accept traffic. The microservices advertise their existence, and make them discoverable. The consumers can then look up the registry to get a view of the service availability and the endpoint locations. The registry, in many cases, is more or less a dump. But the components around the registry make the ecosystem intelligent. There are many subprojects existing under Spring Cloud which support registry and discovery capability. Eureka, ZooKeeper, and Consul are three subprojects implementing the registry capability.
- **Service-to-service calls:** The Spring Cloud Feign subproject under Spring Cloud offers a declarative approach for making RESTful service-to-service calls in a synchronous way. The declarative approach allows applications to work with **POJO (Plain Old Java Object)** interfaces instead of low-level HTTP client APIs. Feign internally uses reactive libraries for communication.

- **Circuit breaker:** The circuit breaker subproject implements the circuit breaker pattern. The circuit breaker breaks the circuit when it encounters failures in the primary service by diverting traffic to another temporary fallback service. It also automatically reconnects back to the primary service when the service is back to normal. It finally provides a monitoring dashboard for monitoring the service state changes. The Spring Cloud Hystrix project and Hystrix Dashboard implement the circuit breaker and the dashboard respectively.
- **Global locks, leadership election and cluster state:** This capability is required for cluster management and coordination when dealing with large deployments. It also offers global locks for various purposes such as sequence generation. The Spring Cloud Cluster project implements these capabilities using Redis, ZooKeeper, and Consul.
- **Security:** Security capability is required for building security for cloud-native distributed systems using externalized authorization providers such as OAuth2. The Spring Cloud Security project implements this capability using customizable authorization and resource servers. It also offers SSO capabilities, which are essential when dealing with many microservices.
- **Big data support:** The big data support capability is a capability that is required for data services and data flows in connection with big data solutions. The Spring Cloud Streams and the Spring Cloud Data Flow projects implement these capabilities. The Spring Cloud Data Flow is the re-engineered version of Spring XD.
- **Distributed tracing:** The distributed tracing capability helps to thread and correlate transitions that are spanned across multiple microservice instances. Spring Cloud Sleuth implements this by providing an abstraction on top of various distributed tracing mechanisms such as Zipkin and HTrace with the support of a 64-bit ID.
- **Distributed messaging:** Spring Cloud Stream provides declarative messaging integration on top of reliable messaging solutions such as Kafka, Redis, and RabbitMQ.
- **Cloud support:** Spring Cloud also provides a set of capabilities that offers various connectors, integration mechanisms, and abstraction on top of different cloud providers such as the Cloud Foundry and AWS.

Spring Cloud and Netflix OSS

Many of the Spring Cloud components which are critical for microservices' deployment came from the **Netflix Open Source Software (Netflix OSS)** center. Netflix is one of the pioneers and early adaptors in the microservices space. In order to manage large scale microservices, engineers at Netflix came up with a number of homegrown tools and techniques for managing their microservices. These are fundamentally crafted to fill some of the software gaps recognized in the AWS platform for managing Netflix services. Later, Netflix open-sourced these components, and made them available under the Netflix OSS platform for public use. These components are extensively used in production systems, and are battle-tested with large scale microservice deployments at Netflix.

Spring Cloud offers higher levels of abstraction for these Netflix OSS components, making it more Spring developer friendly. It also provides a declarative mechanism, well-integrated and aligned with Spring Boot and the Spring framework.

Setting up the environment for BrownField PSS

In this chapter, we will amend the BrownField PSS microservices developed in *Chapter 4, Microservices Evolution – A Case Study*, using Spring Cloud capabilities. We will also examine how to make these services enterprise grade using Spring Cloud components.

Subsequent sections of this chapter will explore how to scale the microservices developed in the previous chapter for cloud scale deployments, using some out-of-the-box capabilities provided by the Spring Cloud project. The rest of this chapter will explore Spring Cloud capabilities such as configuration using the Spring Config server, Ribbon-based service load balancing, service discovery using Eureka, Zuul for API gateway, and finally, Spring Cloud messaging for message-based service interactions. We will demonstrate the capabilities by modifying the BrownField PSS microservices developed in *Chapter 4, Microservices Evolution – A Case Study*.

In order to prepare the environment for this chapter, import and rename (chapter4.* to chapter5.*) projects into a new STS workspace.



The full source code of this chapter is available under the Chapter 5 projects in the code files



Spring Cloud Config

The Spring Cloud Config server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties. The Spring Config server also supports version control of the configuration properties

In the earlier examples with Spring Boot, all configuration parameters were read from a property file packaged inside the project, either `application.properties` or `application.yaml`. This approach is good, since all properties are moved out of code to a property file. However, when microservices are moved from one environment to another, these properties need to undergo changes, which require an application re-build. This is violation of one of the Twelve-Factor application principles, which advocate one-time build and moving of the binaries across environments.

A better approach is to use the concept of profiles. Profiles, as discussed in *Chapter 2, Building Microservices with Spring Boot*, is used for partitioning different properties for different environments. The profile-specific configuration will be named `application-{profile}.properties`. For example, `application-development.properties` represents a property file targeted for the development environment

However, the disadvantage of this approach is that the configurations are statically packaged along with the application. Any changes in the configuration properties require the application to be rebuilt.

There are alternate ways to externalize the configuration properties from the application deployment package. Configurable properties can also be read from an external source in a number of ways:

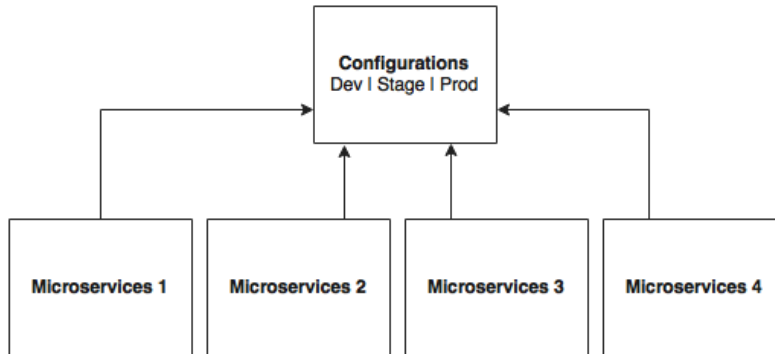
- From an external JNDI server using JNDI namespace (`java:comp/env`)
- Using the Java system properties (`System.getProperties()`) or using the `-D` command line option
- Using the `PropertySource` configuration:

```
@PropertySource("file:${CONF_DIR}/application.properties")
public class ApplicationConfig {
}
```

- Using a command-line parameter pointing a file to an external location:
- ```
java -jar myproject.jar --spring.config.location=
```

JNDI operations are expensive, lack flexibility, have difficulties in replication, and are not version controlled. `System.properties` is not flexible enough for large-scale deployments. The last two options rely on a local or a shared filesystem mounted on the server.

For large scale deployments, a simple yet powerful centralized configuration management solution is required:



As shown in the preceding diagram, all microservices point to a central server to get the required configuration parameters. The microservices then locally cache these parameters to improve performance. The Config server propagates the configuration state changes to all subscribed microservices so that the local cache's state can be updated with the latest changes. The Config server also uses profiles to resolve values specific to an environment.

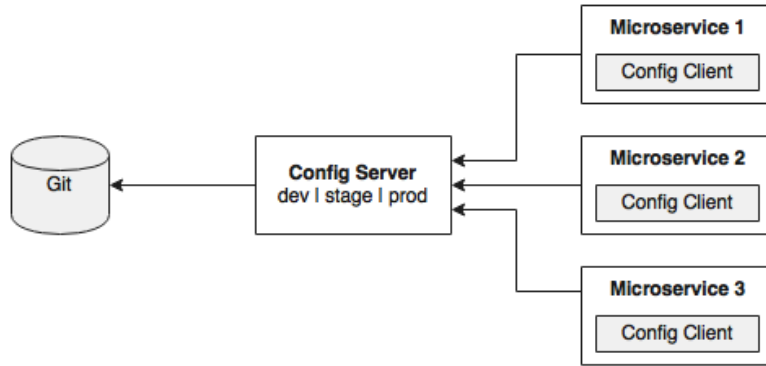
As shown in the following screenshot, there are multiple options available under the Spring Cloud project for building the configuration server. **Config Server**, **Zookeeper Configuration**, and **Consul Configuration** are available as options. However, this chapter will only focus on the Spring Config server implementation.

## Cloud Config

- ☐ Config Client  
spring-cloud-config Client
- ☐ Config Server  
Central management for configuration via a git or svn backend
- ☐ Zookeeper Configuration  
Configuration management with Zookeeper and spring-cloud-zookeeper-config
- ☐ Consul Configuration  
Configuration management with Hashicorp Consul

The Spring Config server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large scale distributed microservice deployments.

The Spring Cloud Config server architecture is shown in the following diagram

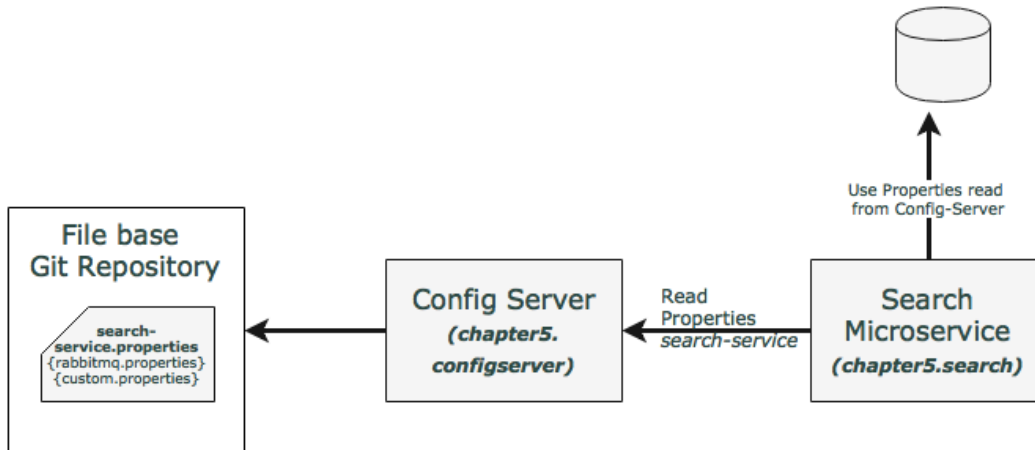


As shown in the preceding diagram, the Config client embedded in the Spring Boot microservices does a configuration lookup from a central configuration server using a simple declarative mechanism, and stores properties into the Spring environment. The configuration properties can be application-level configurations such as a trade limit per day, or infrastructure-related configurations such as server URLs, credentials, and so on.

Unlike Spring Boot, Spring Cloud uses a bootstrap context, which is a parent context of the main application. Bootstrap context is responsible for loading configuration properties from the Config server. The bootstrap context looks for `bootstrap.yaml` or `bootstrap.properties` for loading initial configuration properties. To make this work in a Spring Boot application, rename the `application.*` file to `bootstrap.*`.

## What's next?

The next few sections demonstrate how to use the Config server in a real-world scenario. In order to do this, we will modify our search microservice (chapter5.search) to use the Config server. The following diagram depicts the scenario



In this example, the Search service will read the Config server at startup by passing the service name. In this case, the service name of the search service will be `search-service`. The properties configured for the `search-service` include the RabbitMQ properties as well as a custom property.

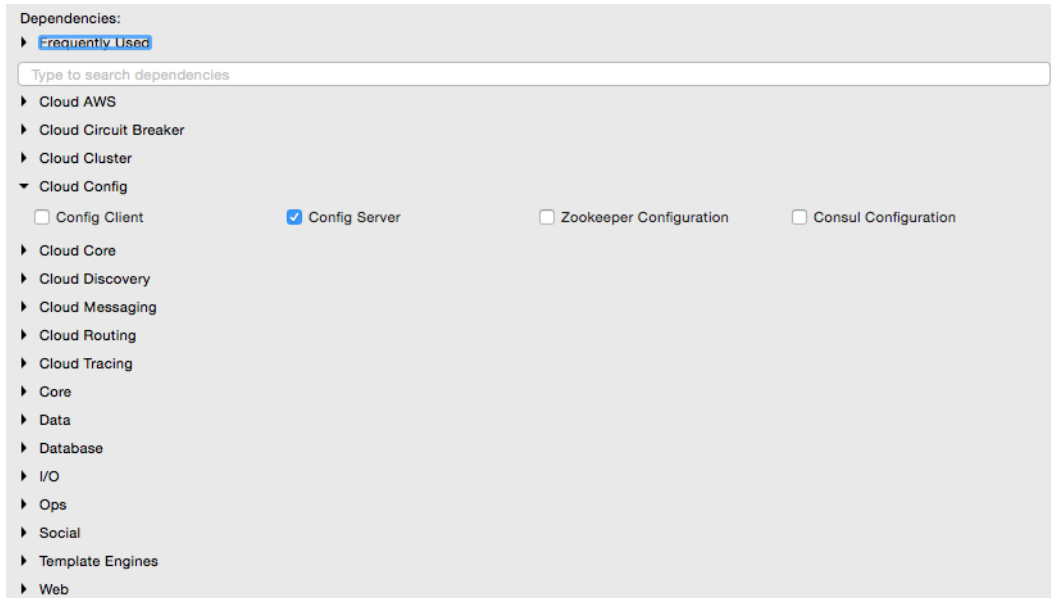


The full source code of this section is available under the `chapter5.configserver` project in the code files

## Setting up the Config server

The following steps need to be followed to create a new Config server using STS

1. Create a new **Spring Starter Project**, and select **Config Server** and **Actuator** as shown in the following diagram:



2. Set up a Git repository. This can be done by pointing to a remote Git configuration repository like the one at <https://github.com/spring-cloud-samples/config-repo>. This URL is an indicative one, a Git repository used by the Spring Cloud examples. We will have to use our own Git repository instead.
3. Alternately, a local filesystem-based Git repository can be used. In a real production scenario, an external Git is recommended. The Config server in this chapter will use a local filesystem-based Git repository for demonstration purposes.
4. Enter the commands listed next to set up a local Git repository:

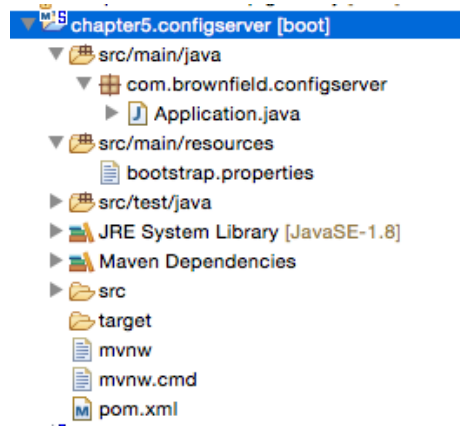
```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo message : helloworld > application.properties
$ git add -A .
$ git commit -m "Added sample application.properties"
```

This code snippet creates a new Git repository on the local filesystem. A property file named `application.properties` with a message property and value `helloworld` is also created.



The file `application.properties` is created for demonstration purposes. We will change this in the subsequent sections.

5. The next step is to change the configuration in the Config server to use the Git repository created in the previous step. In order to do this, rename the file `application.properties` to `bootstrap.properties`:



6. Edit the contents of the new `bootstrap.properties` file to match the following:

```
server.port=8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

Port 8888 is the default port for the Config server. Even without configuring `server.port`, the Config server should bind to 8888. In the Windows environment, an extra `/` is required in the file URL.

7. Optionally, rename the default package of the auto-generated `Application.java` from `com.example` to `com.brownfield.configserver`. Add `@EnableConfigServer` in `Application.java`:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigserverApplication {
```

8. Run the Config server by right-clicking on the project, and running it as a Spring Boot app.
9. Visit `http://localhost:8888/env` to see whether the server is running. If everything is fine, this will list all environment configurations. Note that `/env` is an actuator endpoint.

10. Check `http://localhost:8888/application/default/master` to see the properties specific to `application.properties`, which were added in the earlier step. The browser will display the properties configured in `application.properties`. The browser should display contents similar to the following:

```
{ "name": "application", "profiles": ["default"], "label": "master", "version": "6046fd2ff4fa09d3843767660d963866ffcc7d28", "propertySources": [{ "name": "file:///Users/rvlab /config-repo /application.properties", "source": { "message": "helloworld" } }] }
```

## Understanding the Config server URL

In the previous section, we used `http://localhost:8888/application/default/master` to explore the properties. How do we interpret this URL?

The first element in the URL is the application name. In the given example, the application name should be `application`. The application name is a logical name given to the application, using the `spring.application.name` property in `bootstrap.properties` of the Spring Boot application. Each application must have a unique name. The Config server will use the name to resolve and pick up appropriate properties from the Config server repository. The application name is also sometimes referred to as service ID. If there is an application with the name `myapp`, then there should be a `myapp.properties` in the configuration repository to store all the properties related to that application.

The second part of the URL represents the profile. There can be more than one profile configured within the repository for an application. The profiles can be used in various scenarios. The two common scenarios are segregating different environments such as `Dev`, `Test`, `Stage`, `Prod`, and the like, or segregating server configurations such as `Primary`, `Secondary`, and so on. The first one represents different environments of an application, whereas the second one represents different servers where an application is deployed.

The profile names are logical name that will be used for matching the file name in the repository. The default profile is named `default`. To configure properties for different environments, we have to configure different files as given in the following example. In this example, the first file is for the development environment where the second is for the production environment:

```
application-development.properties
application-production.properties
```

These are accessible using the following URLs respectively:

- `http://localhost:8888/application/development`
- `http://localhost:8888/application/production`

The last part of the URL is the label, and is named `master` by default. The label is an optional Git label that can be used, if required.

In short, the URL is based on the following pattern: `http://localhost:8888/{name}/{profile}/{label}`.

The configuration can also be accessed by ignoring the profile. In the preceding example, all the following three URLs point to the same configuration

- `http://localhost:8888/application/default`
- `http://localhost:8888/application/master`
- `http://localhost:8888/application/default/master`

There is an option to have different Git repositories for different profiles. This makes sense for production systems, since the access to different repositories could be different.

## Accessing the Config Server from clients

In the previous section, a Config server is set up and accessed using a web browser. In this section, the Search microservice will be modified to use the Config server. The Search microservice will act as a Config client.

Follow these steps to use the Config server instead of reading properties from the `application.properties` file:

1. Add the Spring Cloud Config dependency and the actuator (if the actuator is not already in place) to the `pom.xml` file. The actuator is mandatory for refreshing the configuration properties.
2. Since we are modifying the Spring Boot Search microservice from the earlier chapter, we will have to add the following to include the Spring Cloud dependencies. This is not required if the project is created from scratch:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

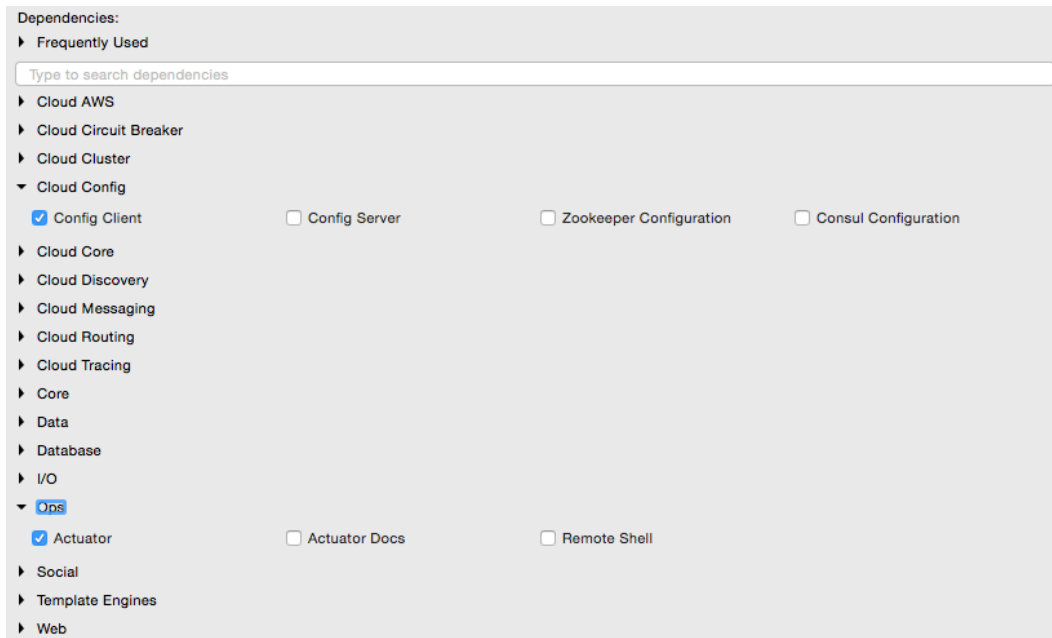
```
<dependencyManagement>
 <dependencies>
 <dependency>
```

```

<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>Brixton.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

```

3. The next screenshot shows the Cloud starter library selection screen. If the application is built from the ground up, select the libraries as shown in the following screenshot:



4. Rename `application.properties` to `bootstrap.properties`, and add an application name and a configuration server URL. The configuration server URL is not mandatory if the Config server is running on the default port (8888) on the local host:

The new `bootstrap.properties` file will look as follows:

```

spring.application.name=search-service
spring.cloud.config.uri=http://localhost:8888

server.port=8090

```

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

search-service is a logical name given to the Search microservice. This will be treated as service ID. The Config server will look for search-service.properties in the repository to resolve the properties.

5. Create a new configuration file for search-service. Create a new search-service.properties under the config-repo folder where the Git repository is created. Note that search-service is the service ID given to the Search microservice in the bootstrap.properties file. Move service-specific properties from bootstrap.properties to the new search-service.properties file. The following properties will be removed from bootstrap.properties, and added to search-service.properties:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

6. In order to demonstrate the centralized configuration of properties and propagation of changes, add a new application-specific property to the property file. We will add originairports.shutdown to temporarily take out an airport from the search. Users will not get any flights when searching for an airport mentioned in the shutdown list:

```
originairports.shutdown=SEA
```

In this example, we will not return any flights when searching with SEA as origin.

7. Commit this new file into the Git repository by executing the following commands:

```
git add -A .
git commit -m "adding new configuration"
```

8. The final search-service.properties file should look as follows

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
originairports.shutdown:SEA
```

9. The `chapter5.search` project's `bootstrap.properties` should look like the following:

```
spring.application.name=search-service
server.port=8090
spring.cloud.config.uri=http://localhost:8888
```

10. Modify the Search microservice code to use the configured parameter, `originairports.shutdown`. A `RefreshScope` annotation has to be added at the class level to allow properties to be refreshed when there is a change. In this case, we are adding a refresh scope to the `SearchRestController` class:

```
@RefreshScope
```

11. Add the following instance variable as a place holder for the new property that is just added in the Config server. The property names in the `search-service.properties` file must match

```
@Value("${originairports.shutdown}")
private String originAirportShutdownList;
```

12. Change the application code to use this property. This is done by modifying the search method as follows:

```
@RequestMapping(value="/get", method =
 RequestMethod.POST)
List<Flight> search(@RequestBody SearchQuery query) {
 logger.info("Input : "+ query);
 if (Arrays.asList(originAirportShutdownList.split(","))
 .contains(query.getOrigin())) {
 logger.info("The origin airport is in shutdown state");
 return new ArrayList<Flight>();
 }
 return searchComponent.search(query);
}
```

The search method is modified to read the parameter `originAirportShutdownList` and see whether the requested origin is in the shutdown list. If there is a match, then instead of proceeding with the actual search, the search method will return an empty flight list.

13. Start the Config server. Then start the Search microservice. Make sure that the RabbitMQ server is running.
14. Modify the `chapter5.website` project to match the `bootstrap.properties` content as follows to utilize the Config server

```
spring.application.name=test-client
server.port=8001
spring.cloud.config.uri=http://localhost:8888
```

15. Change the run method of `CommandLineRunner` in `Application.java` to query SEA as the origin airport:

```
SearchQuery = new SearchQuery("SEA", "SFO", "22-JAN-16");
```

16. Run the `chapter5.website` project. The `CommandLineRunner` will now return an empty flight list. The following message will be printed in the server:

```
The origin airport is in shutdown state
```

## Handling configuration changes

This section will demonstrate how to propagate configuration properties when there is a change:

1. Change the property in the `search-service.properties` file to the following `originairports.shutdown:NYC`

Commit the change in the Git repository. Refresh the Config server URL (<http://localhost:8888/search-service/default>) for this service and see whether the property change is reflected. If everything is fine, we will see the property change. The preceding request will force the Config server to read the property file again from the repository.

2. Rerun the website project again, and observe the `CommandLineRunner` execution. Note that in this case, we are not restarting the Search microservice nor the Config server. The service returns an empty flight list as earlier, and still complains as follows

```
The origin airport is in shutdown state
```

This means the change is not reflected in the Search service, and the service is still working with an old copy of the configuration properties.

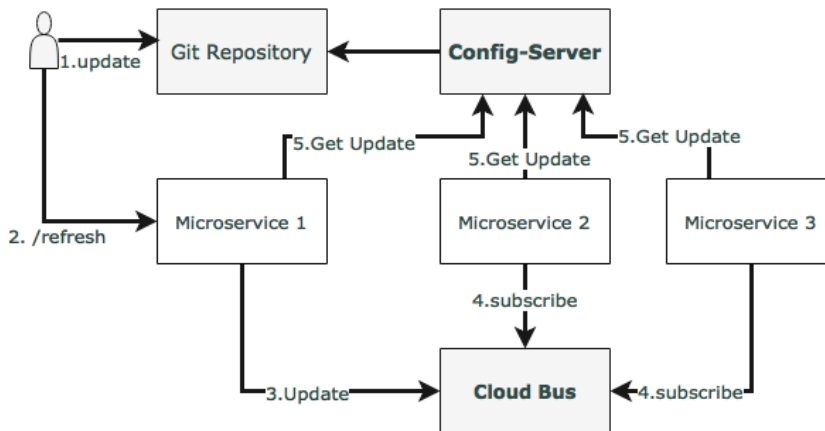
3. In order to force reloading of the configuration properties, call the `/refresh` endpoint of the Search microservice. This is actually the actuator's refresh endpoint. The following command will send an empty POST to the `/refresh` endpoint:

```
curl -d {} localhost:8090/refresh
```

4. Rerun the website project, and observe the `CommandLineRunner` execution. This should return the list of flights that we have requested from SEA. Note that the website project may fail if the Booking service is not up and running. The `/refresh` endpoint will refresh the locally cached configuration properties, and reload fresh values from the Config server.

# Spring Cloud Bus for propagating configuration changes

With the preceding approach, configuration parameters can be changed without restarting the microservices. This is good when there are only one or two instances of the services running. What happens if there are many instances? For example, if there are five instances, then we have to hit `/refresh` against each service instance. This is definitely a cumbersome activity



The Spring Cloud Bus provides a mechanism to refresh configurations across multiple instances without knowing how many instances there are, or their locations. This is particularly handy when there are many service instances of a microservice running or when there are many microservices of different types running. This is done by connecting all service instances through a single message broker. Each instance subscribes for change events, and refreshes its local configuration when required. This refresh is triggered by making a call to any one instance by hitting the `/bus/refresh` endpoint, which then propagates the changes through the cloud bus and the common message broker.

In this example, RabbitMQ is used as the AMQP message broker. Implement this by following the steps documented as follows:

1. Add a new dependency in the `chapter5.search` project's `pom.xml` file to introduce the Cloud Bus dependency:

```

<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>

```



2. The Search microservice also needs connectivity to the RabbitMQ, but this is already provided in `search-service.properties`.
3. Rebuild and restart the Search microservice. In this case, we will run two instances of the Search microservice from a command line, as follows:

```
java -jar -Dserver.port=8090 search-1.0.jar
java -jar -Dserver.port=8091 search-1.0.jar
```

The two instances of the Search service will be now running, one on port 8090 and another one on 8091.

4. Rerun the website project. This is just to make sure that everything is working. The Search service should return one flight at this point
5. Now, update `search-service.properties` with the following value, and commit to Git:

```
originairports.shutdown:SEA
```

6. Run the following command to `/bus/refresh`. Note that we are running a new bus endpoint against one of the instances, 8090 in this case:

```
curl -d {} localhost:8090/bus/refresh
```

7. Immediately, we will see the following message for both instances:

```
Received remote refresh request. Keys refreshed [originairports.
shutdown]
```

The bus endpoint sends a message to the message broker internally, which is eventually consumed by all instances, reloading their property files. Changes can also be applied to a specific application by specifying the application name like so:

```
/bus/refresh?destination=search-service:**
```

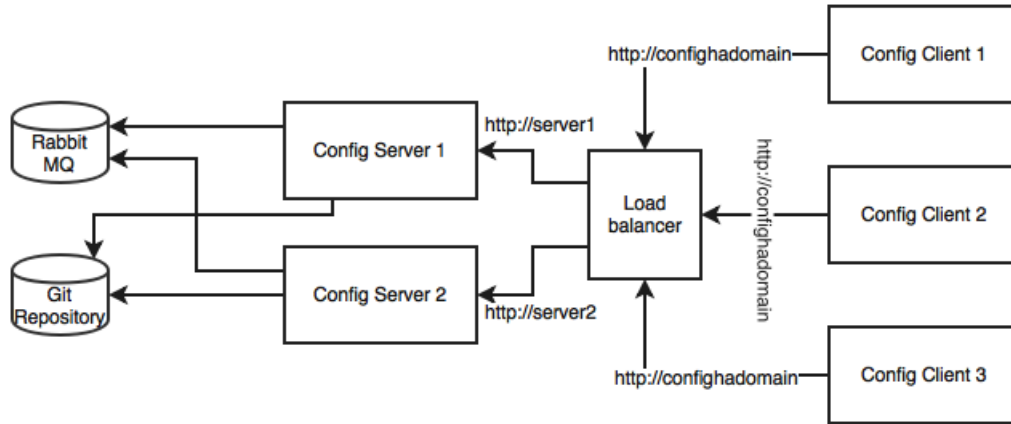
We can also refresh specific properties by setting the property name as a parameter

## Setting up high availability for the Config server

The previous sections explored how to set up the Config server, allowing real-time refresh of configuration properties. However, the Config server is a single point of failure in this architecture.

There are three single points of failure in the default architecture that was established in the previous section. One of them is the availability of the Config server itself, the second one is the Git repository, and the third one is the RabbitMQ server.

The following diagram shows a high availability architecture for the Config server




The architecture mechanisms and rationale are explained as follows:

The Config server requires high availability, since the services won't be able to bootstrap if the Config server is not available. Hence, redundant Config server are required for high availability. However, the applications can continue to run if the Config server is unavailable after the services are bootstrapped. In this case, services will run with the last known configuration state. Hence, the Config server availability is not at the same critical level as the microservices availability.

In order to make the Config server highly available, we need multiple instances of the Config servers. Since the Config server is a stateless HTTP service, multiple instances of configuration servers can be run in parallel. Based on the load on the configuration server, a number of instances have to be adjusted. The `bootstrap.properties` file is not capable of handling more than one server address. Hence, multiple configuration servers should be configured to run behind a load balancer or behind a local DNS with failover and fallback capabilities. The load balancer or DNS server URL will be configured in the microservices' `bootstrap.properties` file. This is with the assumption that the DNS or the load balancer is highly available and capable of handling failovers.


In a production scenario, it is not recommended to use a local file-based Git repository. The configuration server should be typically backed with a highly available Git service. This is possible by either using an external highly available Git service or a highly available internal Git service. SVN can also be considered.

Having said that, an already bootstrapped Config server is always capable of working with a local copy of the configuration. Hence, we need a highly available Git only when the Config server needs to be scaled. Therefore, this too is not as critical as the microservices availability or the Config server availability

 The GitLab example for setting up high availability is available at <https://about.gitlab.com/high-availability/>.

RabbitMQ also has to be configured for high availability. The high availability for RabbitMQ is needed only to push configuration changes dynamically to all instances. Since this is more of an offline controlled activity, it does not really require the same high availability as required by the components.

RabbitMQ high availability can be achieved by either using a cloud service or a locally configured highly available RabbitMQ service

 Setting up high availability for Rabbit MQ is documented at <https://www.rabbitmq.com/ha.html>.

## Monitoring the Config server health

The Config server is nothing but a Spring Boot application, and is, by default, configured with an actuator. Hence, all actuator endpoints are applicable for the Config server. The health of the server can be monitored using the following actuator URL: `http://localhost:8888/health`.

## Config server for configuration files

We may run into scenarios where we need a complete configuration file such as `logback.xml` to be externalized. The Config server provides a mechanism to configure and store such files. This is achievable by using the URL format as follows `/ {name} / {profile} / {label} / {path}`.

The name, profile, and label have the same meanings as explained earlier. The path indicates the file name such as `logback.xml`.

## Completing changes to use the Config server

In order to build this capability to complete BrownField Airline's PSS, we have to make use of the configuration server for all services. All microservices in the examples given in chapter5.\* need to make similar changes to look to the Config server for getting the configuration parameters

The following are a few key change considerations:

- The Fare service URL in the booking component will also be externalized:

```
private static final String FareURL = "/fares";

@Value("${fares-service.url}")
private String fareServiceUrl;

Fare = restTemplate.getForObject(fareServiceUrl+FareURL +"/
get?flightNumber="+record.getFlightNumber()+"&flightDate="+record.
getFlightDate(), Fare.class);
```

As shown in the preceding code snippet, the Fare service URL is fetched through a new property: `fares-service.url`.

- We are not externalizing the queue names used in the Search, Booking, and Check-in services at the moment. Later in this chapter, these will be changed to use Spring Cloud Streams.

## Feign as a declarative REST client

In the Booking microservice, there is a synchronous call to Fare. `RestTemplate` is used for making the synchronous call. When using `RestTemplate`, the URL parameter is constructed programmatically, and data is sent across to the other service. In more complex scenarios, we will have to get to the details of the HTTP APIs provided by `RestTemplate` or even to APIs at a much lower level.

Feign is a Spring Cloud Netflix library for providing a higher level of abstraction over REST-based service calls. Spring Cloud Feign works on a declarative principle. When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client. The developer need not worry about the implementation of this interface. This will be dynamically provisioned by Spring at runtime. With this declarative approach, developers need not get into the details of the HTTP level APIs provided by `RestTemplate`.

The following code snippet is the existing code in the Booking microservice for calling the Fare service:

```
Fare fare = restTemplate.getForObject(FareURL + "/"
get?flightNumber="+record.getFlightNumber()+"&flightDate="+record.
getFlightDate(), Fare.class);
```

In order to use Feign, first we need to change the `pom.xml` file to include the Feign dependency as follows:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
```

For a new Spring Starter project, **Feign** can be selected from the starter library selection screen, or from <http://start.spring.io/>. This is available under **Cloud Routing** as shown in the following screenshot:

### Cloud Routing

- ☐ Zuul  
Intelligent and programmable routing with spring-cloud-netflix Zuul
- ☐ Ribbon  
Client side load balancing with spring-cloud-netflix and Ribbon
- ☐ Feign  
Declarative REST clients with spring-cloud-netflix Feign

The next step is to create a new `FareServiceProxy` interface. This will act as a proxy interface of the actual Fare service:

```
@FeignClient(name="fares-proxy", url="localhost:8080/fares")
public interface FareServiceProxy {
 @RequestMapping(value = "/get", method=RequestMethod.GET)
 Fare getFare(@RequestParam(value="flightNumber") String
 flightNumber, @RequestParam(value="flightDate") String
 flightDate);
}
```

The `FareServiceProxy` interface has a `@FeignClient` annotation. This annotation tells Spring to create a REST client based on the interface provided. The value could be a service ID or a logical name. The `url` indicates the actual URL where the target service is running. Either name or value is mandatory. In this case, since we have `url`, the `name` attribute is irrelevant.

Use this service proxy to call the Fare service. In the Booking microservice, we have to tell Spring that Feign clients exist in the Spring Boot application, which are to be scanned and discovered. This will be done by adding `@EnableFeignClients` at the class level of `BookingComponent`. Optionally, we can also give the package names to scan.

Change `BookingComponent`, and make changes to the calling part. This is as simple as calling another Java interface:

```
Fare = fareServiceProxy.getFare(record.getFlightNumber(), record.
getFlightDate());
```

Rerun the Booking microservice to see the effect.

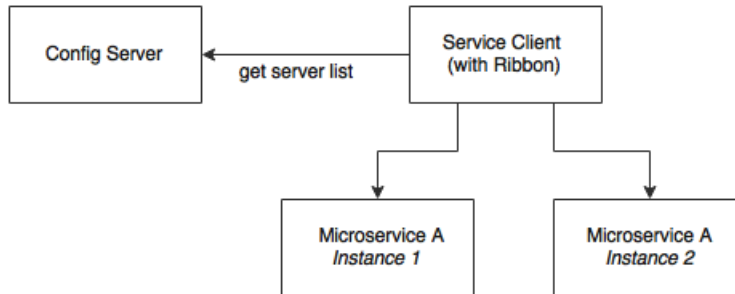
The URL of the Fare service in the `FareServiceProxy` interface is hardcoded:  
`url="localhost:8080/fares".`

For the time being, we will keep it like this, but we are going to change this later in this chapter.

## Ribbon for load balancing

In the previous setup, we were always running with a single instance of the microservice. The URL is hardcoded both in client as well as in the service-to-service calls. In the real world, this is not a recommended approach, since there could be more than one service instance. If there are multiple instances, then ideally, we should use a load balancer or a local DNS server to abstract the actual instance locations, and configure an alias name or the load balancer address in the clients. The load balancer then receives the alias name, and resolves it with one of the available instances. With this approach, we can configure as many instances behind a load balancer. It also helps us to handle server failures transparent to the client.

This is achievable with Spring Cloud Netflix Ribbon. Ribbon is a client-side load balancer which can do round-robin load balancing across a set of servers. There could be other load balancing algorithms possible with the Ribbon library. Spring Cloud offers a declarative way to configure and use the Ribbon client



As shown in the preceding diagram, the Ribbon client looks for the Config server to get the list of available microservice instances, and, by default, applies a round-robin load balancing algorithm.

In order to use the Ribbon client, we will have to add the following dependency to the `pom.xml` file

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

In case of development from ground up, this can be selected from the Spring Starter libraries, or from <http://start.spring.io/>. Ribbon is available under **Cloud Routing**:

## Cloud Routing

- ☐ Zuul  
Intelligent and programmable routing with spring-cloud-netflix Zuul
- ☐ Ribbon  
Client side load balancing with spring-cloud-netflix and Ribbon
- ☐ Feign  
Declarative REST clients with spring-cloud-netflix Feign

Update the Booking microservice configuration file `booking-service.properties`, to include a new property to keep the list of the Fare microservices:

```
fares-proxy.ribbon.listOfServers=localhost:8080,localhost:8081
```

Going back and editing the `FareServiceProxy` class created in the previous section to use the Ribbon client, we note that the value of the `@RequestMapping` annotations is changed from `/get` to `/fares/get` so that we can move the host name and port to the configuration easily

```
@FeignClient(name="fares-proxy")
@RibbonClient(name="fares")
public interface FareServiceProxy {
 @RequestMapping(value = "/fares/get", method=RequestMethod.GET)
```

We can now run two instances of the Fares microservices. Start one of them on 8080, and the other one on 8081:

```
java -jar -Dserver.port=8080 fares-1.0.jar
java -jar -Dserver.port=8081 fares-1.0.jar
```

Run the Booking microservice. When the Booking microservice is bootstrapped, the `CommandLineRunner` automatically inserts one booking record. This will go to the first server

When running the website project, it calls the Booking service. This request will go to the second server.

On the Booking service, we see the following trace, which says there are two servers enlisted:

```
DynamicServerListLoadBalancer:{NFLoadBalancer:name=fares-proxy,current
list of Servers=[localhost:8080, localhost:8081],Load balancer stats=Zone
stats: {unknown=[Zone:unknown; Instance count:2; Active connections
count: 0; Circuit breaker tripped count: 0; Active connections per
server: 0.0;]}
},
```



## Eureka for registration and discovery

So far, we have achieved externalizing configuration parameters as well as load balancing across many service instances.

Ribbon-based load balancing is sufficient for most of the microservices requirements. However, this approach falls short in a couple of scenarios:

- If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and the associated servers. It is not easy to predict and preconfigure the server URLs in a configuration file.
- When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment.
- In the cloud deployment scenarios, IP addresses are not predictable, and will be difficult to statically configure in a file. We will have to update the configuration file every time there is a change in address.

The Ribbon approach partially addresses this issue. With Ribbon, we can dynamically change the service instances, but whenever we add new service instances or shut down instances, we will have to manually update the Config server. Though the configuration changes will be automatically propagated to all required instances, the manual configuration changes will not work with large scale deployments. When managing large deployments, automation, wherever possible, is paramount.

To fix this gap, the microservices should self-manage their life cycle by dynamically registering service availability, and provision automated discovery for consumers.

## Understanding dynamic service registration and discovery

Dynamic registration is primarily from the service provider's point of view. With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the service registry. The registry always keeps up-to-date information of the services available, as well as their metadata.

Dynamic discovery is applicable from the service consumer's point of view. Dynamic discovery is where clients look for the service registry to get the current state of the services topology, and then invoke the services accordingly. In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.

The clients may keep a local cache of the registry data for faster access. Some registry implementations allow clients to keep a watch on the items they are interested in. In this approach, the state changes in the registry server will be propagated to the interested parties to avoid using stale data.

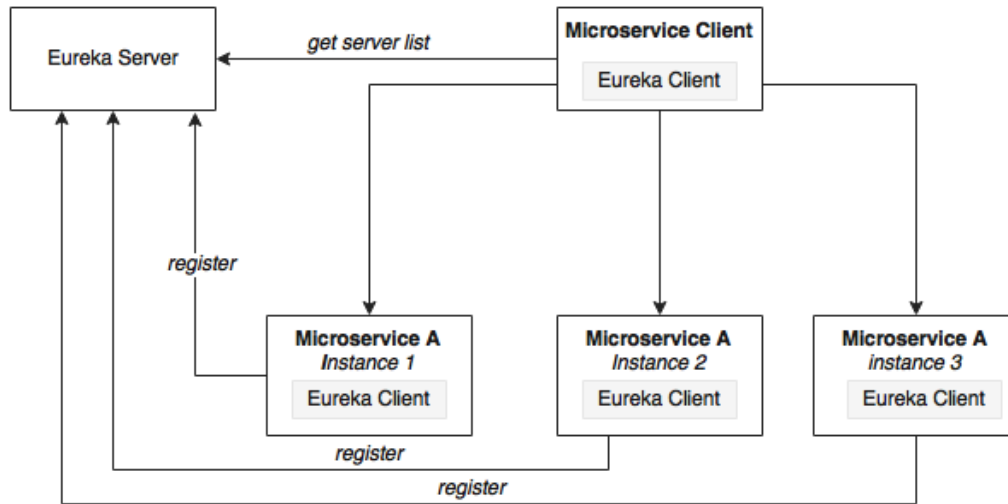
There are a number of options available for dynamic service registration and discovery. Netflix Eureka, ZooKeeper, and Consul are available as part of Spring Cloud, as shown in the <http://start.spring.io/> screenshot given next. Etcd is another service registry available outside of Spring Cloud to achieve dynamic service registration and discovery. In this chapter, we will focus on the Eureka implementation:

## Cloud Discovery

- ☐ Eureka Discovery  
Service discovery using spring-cloud-netflix and Eureka
- ☐ Eureka Server  
spring-cloud-netflix Eureka Server
- ☐ Zookeeper Discovery  
Service discovery with Zookeeper and spring-cloud-zookeeper-discovery
- ☐ Cloud Foundry Discovery  
Service discovery with Cloud Foundry
- ☐ Consul Discovery  
Service discovery with Hashicorp Consul

## Understanding Eureka

Spring Cloud Eureka also comes from Netflix OSS. The Spring Cloud project provides a Spring-friendly declarative approach for integrating Eureka with Spring-based applications. Eureka is primarily used for self-registration, dynamic discovery, and load balancing. Eureka uses Ribbon for load balancing internally:



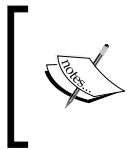
As shown in the preceding diagram, Eureka consists of a server component and a client-side component. The server component is the registry in which all microservices register their availability. The registration typically includes service identity and its URLs. The microservices use the Eureka client for registering their availability. The consuming components will also use the Eureka client for discovering the service instances.

When a microservice is bootstrapped, it reaches out to the Eureka server, and advertises its existence with the binding information. Once registered, the service endpoint sends ping requests to the registry every 30 seconds to renew its lease. If a service endpoint cannot renew its lease in a few attempts, that service endpoint will be taken out of the service registry. The registry information will be replicated to all Eureka clients so that the clients have to go to the remote Eureka server for each and every request. Eureka clients fetch the registry information from the server, and cache it locally. After that, the clients use that information to find other services. This information is updated periodically (every 30 seconds) by getting the delta updates between the last fetch cycle and the current one.

When a client wants to contact a microservice endpoint, the Eureka client provides a list of currently available services based on the requested service ID. The Eureka server is zone aware. Zone information can also be supplied when registering a service. When a client requests for a services instance, the Eureka service tries to find the service running in the same zone. The Ribbon client then load balances across these available service instances supplied by the Eureka client. The communication between the Eureka client and the server is done using REST and JSON.

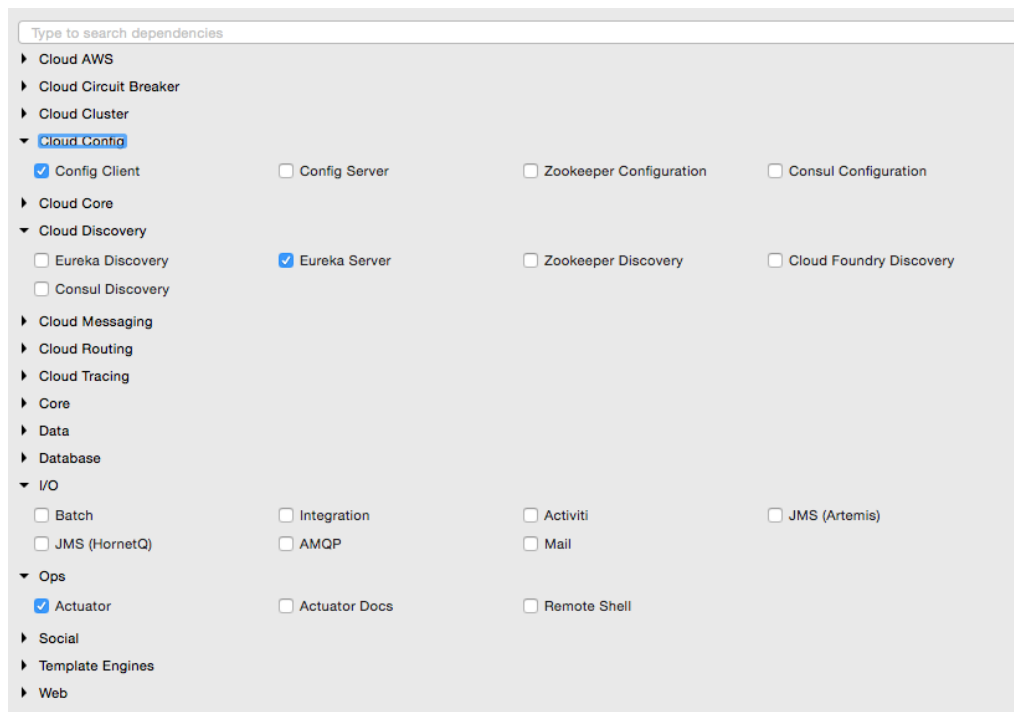
## Setting up the Eureka server

In this section, we will run through the steps required for setting up the Eureka server.

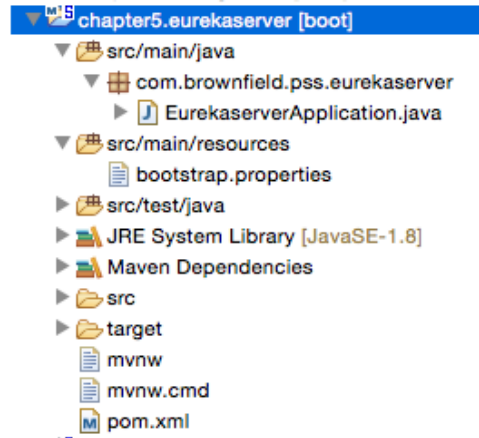


The full source code of this section is available under the `chapter5.eurekaserver` project in the code files. Note that the Eureka server registration and refresh cycles take up to 30 seconds. Hence, when running services and clients, wait for 40-50 seconds.

1. Start a new Spring Starter project, and select **Config Client**, **Eureka Server**, and **Actuator**:



The project structure of the Eureka server is shown in the following image:



Note that the main application is named `EurekaserverApplication.java`.

2. Rename `application.properties` to `bootstrap.properties` since this is using the Config server. As we did earlier, configure the details of the Config server in the `bootstrap.properties` file so that it can locate the Config server instance. The `bootstrap.properties` file will look as follows

```
spring.application.name=eureka-server1
server.port:8761
spring.cloud.config.uri=http://localhost:8888
```

The Eureka server can be set up in a standalone mode or in a clustered mode. We will start with the standalone mode. By default, the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability. The client component is responsible for synchronizing state from the other Eureka servers. The Eureka client is taken to its peers by configuring the `eureka.client.serviceUrl.defaultZone` property.

In the standalone mode, we point `eureka.client.serviceUrl.defaultZone` back to the same standalone instance. Later we will see how we can run Eureka servers in a clustered mode.

3. Create a `eureka-server1.properties` file, and update it in the Git repository. `eureka-server1` is the name of the application given in the application's `bootstrap.properties` file in the previous step. As shown in the following code, `serviceUrl` points back to the same server. Once the following properties are added, commit the file to the Git repository

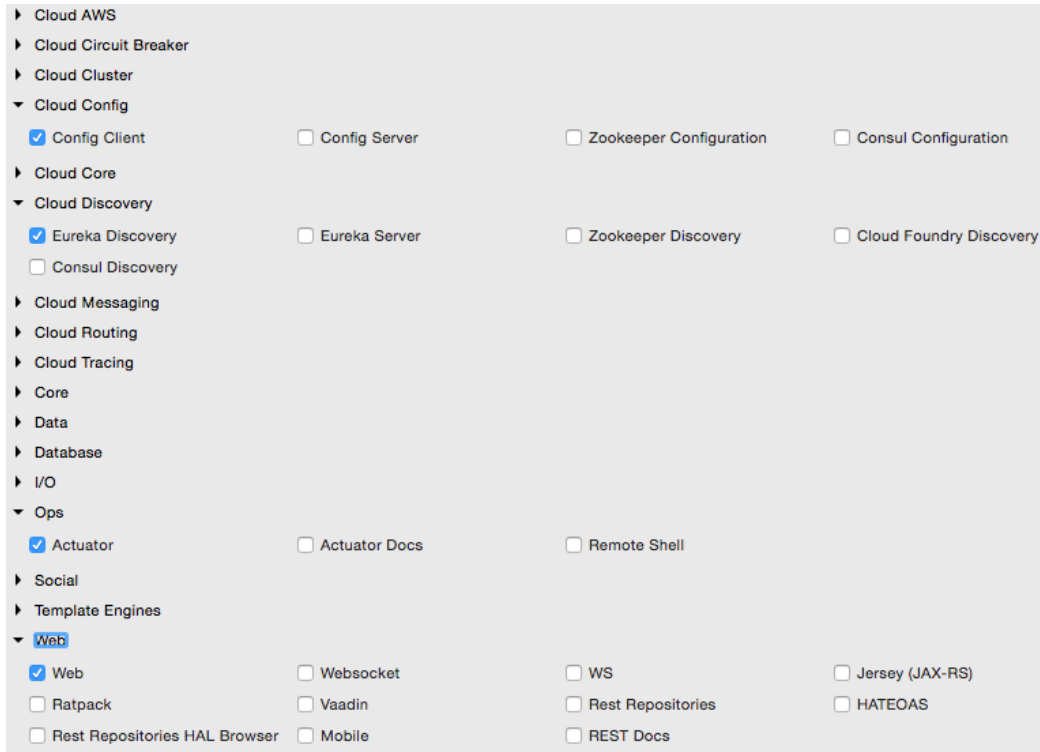
```
spring.application.name=eureka-server1
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

4. Change the default `Application.java`. In this example, the package is also renamed as `com.brownfield.pss.eurekaserver`, and the class name changed to `EurekaserverApplication`. In `EurekaserverApplication`, add `@EnableEurekaServer`:

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaserverApplication {
```

5. We are now ready to start the Eureka server. Ensure that the Config server is also started. Right-click on the application and then choose **Run As | Spring Boot App**. Once the application is started, open `http://localhost:8761` in a browser to see the Eureka console.
6. In the console, note that there is no instance registered under **Instances currently registered with Eureka**. Since no services have been started with the Eureka client enabled, the list is empty at this point.

7. Making a few changes to our microservice will enable dynamic registration and discovery using the Eureka service. To do this, first we have to add the Eureka dependencies to the `pom.xml` file. If the services are being built up fresh using the Spring Starter project, then select **Config Client**, **Actuator**, **Web** as well as **Eureka discovery** client as follows:



8. Since we are modifying our microservices, add the following additional dependency to all microservices in their `pom.xml` files

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

9. The following property has to be added to all microservices in their respective configuration files under `config-repo`. This will help the microservices to connect to the Eureka server. Commit to Git once updates are completed:

```
eureka.client.serviceUrl.defaultZone: http://localhost:8761/
eureka/
```

10. Add `@EnableDiscoveryClient` to all microservices in their respective Spring Boot main classes. This asks Spring Boot to register these services at start up to advertise their availability.
11. Start all servers except Booking. Since we are using the Ribbon client on the Booking service, the behavior could be different when we add the Eureka client in the class path. We will fix this soon
12. Going to the Eureka URL (<http://localhost:8761>), you can see that all three instances are up and running:

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

Time to fix the issue with Booking. We will remove our earlier Ribbon client, and use Eureka instead. Eureka internally uses Ribbon for load balancing. Hence, the load balancing behavior will not change.

13. Remove the following dependency:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
```

14. Also remove the `@RibbonClient (name="fares")` annotation from the `FareServiceProxy` class.
15. Update `@FeignClient (name="fares-service")` to match the actual Fare microservices' service ID. In this case, `fare-service` is the service ID configured in the Fare microservices' `bootstrap.properties`. This is the name that the Eureka discovery client sends to the Eureka server. The service ID will be used as a key for the services registered in the Eureka server.
16. Also remove the list of servers from the `booking-service.properties` file. With Eureka, we are going to dynamically discover this list from the Eureka server:

```
fares-proxy.ribbon.listOfServers=localhost:8080, localhost:8081
```



17. Start the Booking service. You will see that `CommandLineRunner` successfully created a booking, which involves calling the Fare services using the Eureka discovery mechanism. Go back to the URL to see all the registered services:

#### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BOOK-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:book-service:8060
CHECKIN-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:checkin-service:8070
FARES-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:fares-service:8080
SEARCH-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.102:search-service:8090

18. Change the website project's `bootstrap.properties` file to make use of Eureka rather than connecting directly to the service instances. We will not use the Feign client in this case. Instead, for demonstration purposes, we will use the load balanced `RestTemplate`. Commit these changes to the Git repository:

```
spring.application.name=test-client
eureka.client.serviceUrl.defaultZone: http://localhost:8761/
eureka/
```

19. Add `@EnableDiscoveryClient` to the `Application` class to make the client Eureka-aware.
20. Edit both `Application.java` as well as `BrownFieldSiteController.java`. Add three `RestTemplate` instances. This time, we annotate them with `@Loadbalanced` to ensure that we use the load balancing features using Eureka and Ribbon. `RestTemplate` cannot be automatically injected. Hence, we have to provide a configuration entry as follows

```
@Configuration
class AppConfiguration {
 @LoadBalanced
 @Bean
 RestTemplate restTemplate() {
 return new RestTemplate();
 }
}

@Autowired
RestTemplate searchClient;

@Autowired
RestTemplate bookingClient;

@Autowired
RestTemplate checkInClient;
```

21. We use these `RestTemplate` instances to call the microservices. Replace the hardcoded URLs with service IDs that are registered in the Eureka server. In the following code, we use the service names `search-service`, `book-service`, and `checkin-service` instead of explicit host names and ports:

```
Flight[] flights = searchClient.postForObject("http://search-service/search/get", searchQuery, Flight[].class);

long bookingId = bookingClient.postForObject("http://book-service/booking/create", booking, long.class);

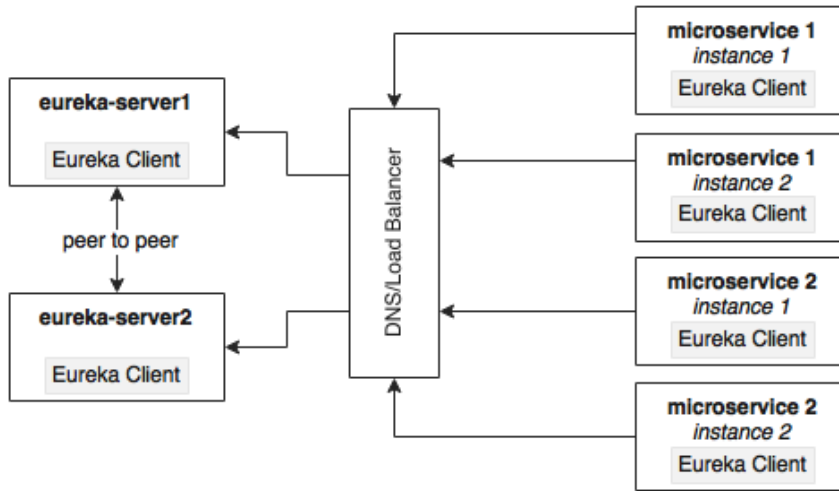
long checkinId = checkInClient.postForObject("http://checkin-service/checkin/create", checkIn, long.class);
```
22. We are now ready to run the client. Run the website project. If everything is fine, the website project's `CommandLineRunner` will successfully perform search, booking, and check-in. The same can also be tested using the browser by pointing the browser to `http://localhost:8001`.

## High availability for Eureka

In the previous example, there was only one Eureka server in standalone mode. This is not good enough for a real production system.

The Eureka client connects to the server, fetches registry information, and stores it locally in a cache. The client always works with this local cache. The Eureka client checks the server periodically for any state changes. In the case of a state change, it downloads the changes from the server, and updates the cache. If the Eureka server is not reachable, then the Eureka clients can still work with the last-known state of the servers based on the data available in the client cache. However, this could lead to stale state issues quickly.

This section will explore the high availability for the Eureka server. The high availability architecture is shown in the following diagram:



The Eureka server is built with a peer-to-peer data synchronization mechanism. The runtime state information is not stored in a database, but managed using an in-memory cache. The high availability implementation favors availability and partition tolerance in the CAP theorem, leaving out consistency. Since the Eureka server instances are synchronized with each other using an asynchronous mechanism, the states may not always match between server instances. The peer-to-peer synchronization is done by pointing `serviceUrls` to each other. If there is more than one Eureka server, each one has to be connected to at least one of the peer servers. Since the state is replicated across all peers, Eureka clients can connect to any one of the available Eureka servers.

The best way to achieve high availability for Eureka is to cluster multiple Eureka servers, and run them behind a load balancer or a local DNS. The clients always connect to the server using the DNS/load balancer. At runtime, the load balancer takes care of selecting the appropriate servers. This load balancer address will be provided to the Eureka clients.

This section will showcase how to run two Eureka servers in a cluster for high availability. For this, define two property files `eureka-server1` and `eureka-server2`. These are peer servers; if one fails, the other one will take over. Each of these servers will also act as a client for the other so that they can sync their states. Two property files are defined in the following snippet. Upload and commit these properties to the Git repository.

The client URLs point to each other, forming a peer network as shown in the following configuration

**eureka-server1.properties**

```
eureka.client.serviceUrl.defaultZone:http://localhost:8762/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

**eureka-server2.properties**

```
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

Update the `bootstrap.properties` file of Eureka, and change the application name to `eureka`. Since we are using two profiles, based on the active profile supplied a startup, the Config server will look for either `eureka-server1` or `eureka-server2`:

```
spring.application.name=eureka
spring.cloud.config.uri=http://localhost:8888
```

Start two instances of the Eureka servers, `server1` on 8761 and `server2` on 8762:

```
java -jar -Dserver.port=8761 -Dspring.profiles.active=server1 demo-0.0.1-SNAPSHOT.jar
```

```
java -jar -Dserver.port=8762 -Dspring.profiles.active=server2 demo-0.0.1-SNAPSHOT.jar
```

All our services still point to the first server, `server1`. Open both the browser windows: `http://localhost:8761` and `http://localhost:8762`.

Start all microservices. The one which opened 8761 will immediately reflect the changes, whereas the other one will take 30 seconds for reflecting the states. Since both the servers are in a cluster, the state is synchronized between these two servers. If we keep these servers behind a load balancer/DNS, then the client will always connect to one of the available servers.

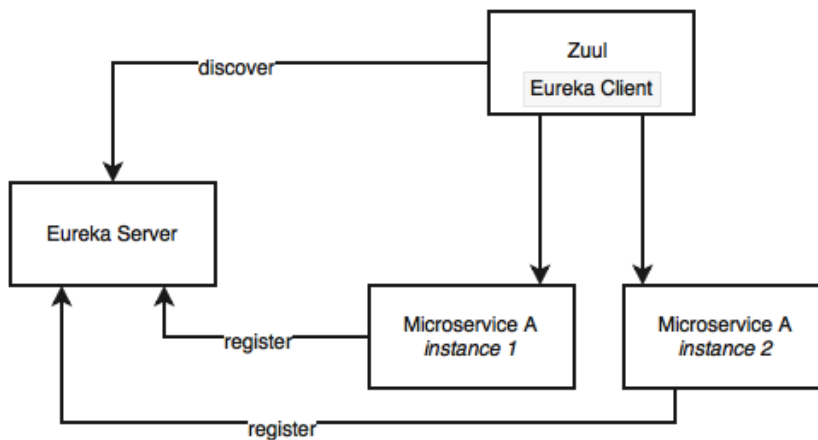
After completing this exercise, switch back to the standalone mode for the remaining exercises.

## Zuul proxy as the API gateway

In most microservice implementations, internal microservice endpoints are not exposed outside. They are kept as private services. A set of public services will be exposed to the clients using an API gateway. There are many reasons to do this:

- Only a selected set of microservices are required by the clients.
- If there are client-specific policies to be applied, it is easy to apply them in a single place rather than in multiple places. An example of such a scenario is the cross-origin access policy.
- It is hard to implement client-specific transformations at the service endpoint.
- If there is data aggregation required, especially to avoid multiple client calls in a bandwidth-restricted environment, then a gateway is required in the middle.

Zuul is a simple gateway service or edge service that suits these situations well. Zuul also comes from the Netflix family of microservice products. Unlike many enterprise API gateway products, Zuul provides complete control for the developers to configure or program based on specific requirement




The Zuul proxy internally uses the Eureka server for service discovery, and Ribbon for load balancing between service instances.

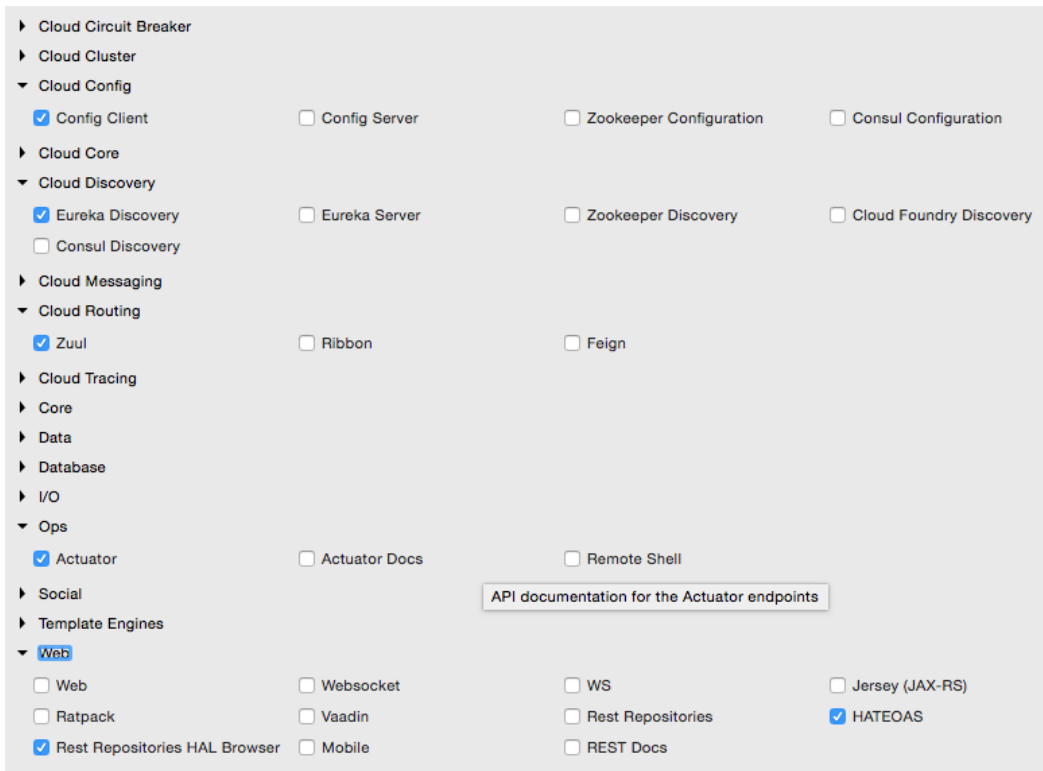
The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on. In simple terms, we can consider Zuul a reverse proxy service. With Zuul, we can even change the behaviors of the underlying services by overriding them at the API layer.

# Setting up Zuul

Unlike the Eureka server and the Config server, in typical deployments, Zuul is specific to a microservice. However, there are deployments in which one API gateway covers many microservices. In this case, we are going to add Zuul for each of our microservices: Search, Booking, Fare, and Check-in:

 The full source code of this section is available under the `chapter5.*-apigateway` project in the code files

1. Convert the microservices one by one. Start with Search API Gateway. Create a new Spring Starter project, and select **Zuul**, **Config Client**, **Actuator**, and **Eureka Discovery**:



The screenshot shows the Spring Initializr project configuration page. The following dependencies are selected (checked):

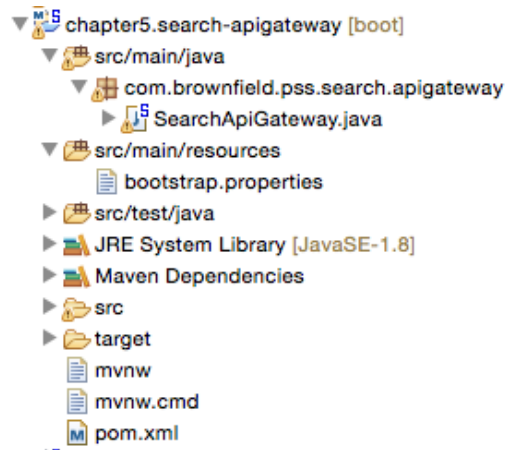
- Cloud Config**
  - ☒ Config Client
- Cloud Discovery**
  - ☒ Eureka Discovery
- Cloud Routing**
  - ☒ Zuul
- Ops**
  - ☒ Actuator
- Web**
  - ☒ Rest Repositories HAL Browser
  - ☒ HATEOAS

The following dependencies are not selected (unchecked):

- Cloud Config**
  - ☐ Config Server
  - ☐ Zookeeper Configuration
  - ☐ Consul Configuration
- Cloud Discovery**
  - ☐ Eureka Server
  - ☐ Zookeeper Discovery
  - ☐ Cloud Foundry Discovery
  - ☐ Consul Discovery
- Cloud Routing**
  - ☐ Ribbon
  - ☐ Feign
- Ops**
  - ☐ Actuator Docs
  - ☐ Remote Shell
- Web**
  - ☐ Web
  - ☐ Ratpack
  - ☐ Websocket
  - ☐ Vaadin
  - ☐ WS
  - ☐ Rest Repositories
  - ☐ REST Docs
  - ☐ Jersey (JAX-RS)
  - ☐ Mobile

A tooltip is visible over the **Actuator** section, displaying the text: "API documentation for the Actuator endpoints".

The project structure for `search-apigateway` is shown in the following diagram:



2. The next step is to integrate the API gateway with Eureka and the Config server. Create a `search-apigateway.properties` file with the contents given next, and commit to the Git repository.

This configuration also sets a rule on how to forward traffic. In this case, any request coming on the `/api` endpoint of the API gateway should be sent to `search-service`:

```
spring.application.name=search-apigateway
zuul.routes.search-apigateway.serviceId=search-service
zuul.routes.search-apigateway.path=/api/**
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
```

`search-service` is the service ID of the Search service, and it will be resolved using the Eureka server.

3. Update the `bootstrap.properties` file of `search-apigateway` as follows. There is nothing new in this configuration – a name to the service, the port, and the Config server URL

```
spring.application.name=search-apigateway
server.port=8095
spring.cloud.config.uri=http://localhost:8888
```

4. Edit `Application.java`. In this case, the package name and the class name are also changed to `com.brownfield.pss.search.apigateway` and `SearchApiGateway` respectively. Also add `@EnableZuulProxy` to tell Spring Boot that this is a Zuul proxy:

```
@EnableZuulProxy
@EnableDiscoveryClient
@SpringBootApplication
public class SearchApiGateway {
```

5. Run this as a Spring Boot app. Before that, ensure that the Config server, the Eureka server, and the Search microservice are running.
6. Change the website project's `CommandLineRunner` as well as `BrownFieldSiteController` to make use of the API gateway:

```
Flight[] flights = searchClient.postForObject("http://search-
apigateway/api/search/get", searchQuery, Flight[].class);
```

In this case, the Zuul proxy acts as a reverse proxy which proxies all microservice endpoints to consumers. In the preceding example, the Zuul proxy does not add much value, as we just pass through the incoming requests to the corresponding backend service.

Zuul is particularly useful when we have one or more requirements like the following:

- Enforcing authentication and other security policies at the gateway instead of doing that on every microservice endpoint. The gateway can handle security policies, token handling, and so on before passing the request to the relevant services behind. It can also do basic rejections based on some business policies such as blocking requests coming from certain black-listed users.
- Business insights and monitoring can be implemented at the gateway level. Collect real-time statistical data, and push it to an external system for analysis. This will be handy as we can do this at one place rather than applying it across many microservices.
- API gateways are useful in scenarios where dynamic routing is required based on fine-grained controls. For example, send requests to different service instances based on business specific values such as "origin country". Another example is all requests coming from a region to be sent to one group of service instances. Yet another example is all requests requesting for a particular product have to be routed to a group of service instances.



- Handling the load shredding and throttling requirements is another scenario where API gateways are useful. This is when we have to control load based on set thresholds such as number of requests in a day. For example, control requests coming from a low-value third party online channel.
- The Zuul gateway is useful for fine-grained load balancing scenarios. The Zuul, Eureka client, and Ribbon together provide fine-grained controls over the load balancing requirements. Since the Zuul implementation is nothing but another Spring Boot application, the developer has full control over the load balancing.
- The Zuul gateway is also useful in scenarios where data aggregation requirements are in place. If the consumer wants higher level coarse-grained services, then the gateway can internally aggregate data by calling more than one service on behalf of the client. This is particularly applicable when the clients are working in low bandwidth environments.

Zuul also provides a number of filters. These filters are classified as pre filter routing filters, post filters, and error filters. As the names indicate, these are applied at different stages of the life cycle of a service call. Zuul also provides an option for developers to write custom filters. In order to write a custom filter, extend from the abstract `ZuulFilter`, and implement the following methods:

```
public class CustomZuulFilter extends ZuulFilter{
 public Object run() {}
 public boolean shouldFilter() {}
 public int filterOrder() {}
 public String filterType() {}
}
```

Once a custom filter is implemented, add that class to the main context. In our example case, add this to the `SearchApiGateway` class as follows:

```
@Bean
public CustomZuulFilter customFilter() {
 return new CustomZuulFilter();
}
```

As mentioned earlier, the Zuul proxy is a Spring Boot service. We can customize the gateway programmatically in the way we want. As shown in the following code, we can add custom endpoints to the gateway, which, in turn, can call the backend services:

```
@RestController
class SearchAPIGatewayController {

 @RequestMapping("/")
 String greet (HttpServletRequest req) {
```

```
 return "<H1>Search Gateway Powered By Zuul</H1>";
 }
}
```

In the preceding case, it just adds a new endpoint, and returns a value from the gateway. We can further use `@Loadbalanced RestTemplate` to call a backend service. Since we have full control, we can do transformations, data aggregation, and so on. We can also use the Eureka APIs to get the server list, and implement completely independent load-balancing or traffic-shaping mechanisms instead of the out-of-the-box load balancing features provided by Ribbon.

## High availability of Zuul

Zuul is just a stateless service with an HTTP endpoint, hence, we can have as many Zuul instances as we need. There is no affinity or stickiness required. However, the availability of Zuul is extremely critical as all traffic from the consumer to the provider flows through the Zuul proxy. However, the elastic scaling requirements are not as critical as the backend microservices where all the heavy lifting happens.

The high availability architecture of Zuul is determined by the scenario in which we are using Zuul. The typical usage scenarios are:

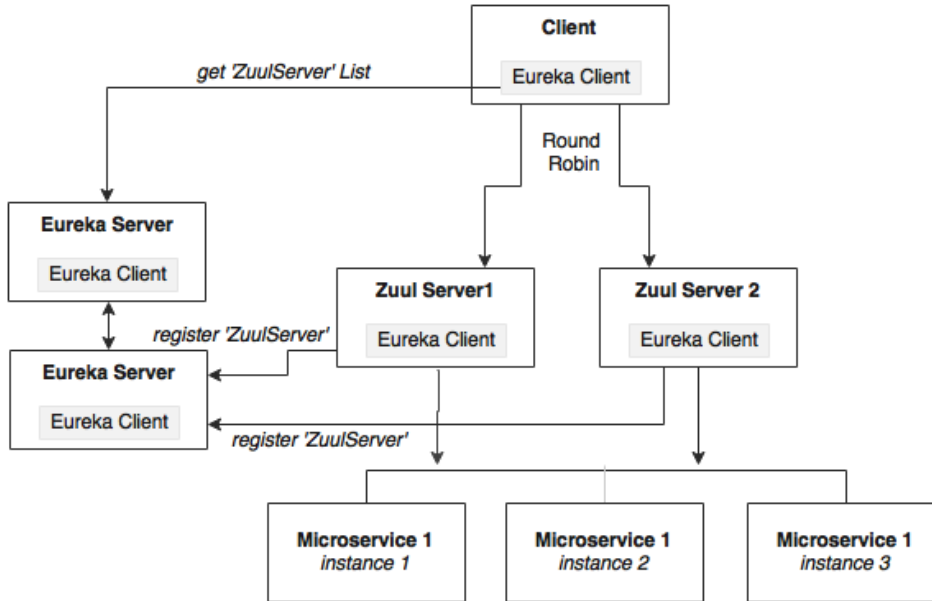
- When a client-side JavaScript MVC such as AngularJS accesses Zuul services from a remote browser.
- Another microservice or non-microservice accesses services via Zuul

In some cases, the client may not have the capabilities to use the Eureka client libraries, for example, a legacy application written on PL/SQL. In some cases, organization policies do not allow Internet clients to handle client-side load balancing. In the case of browser-based clients, there are third-party Eureka JavaScript libraries available.

It all boils down to whether the client is using Eureka client libraries or not. Based on this, there are two ways we can set up Zuul for high availability.

## High availability of Zuul when the client is also a Eureka client

In this case, since the client is also another Eureka client, Zuul can be configured just like other microservices. Zuul registers itself to Eureka with a service ID. The clients then use Eureka and the service ID to resolve Zuul instances:



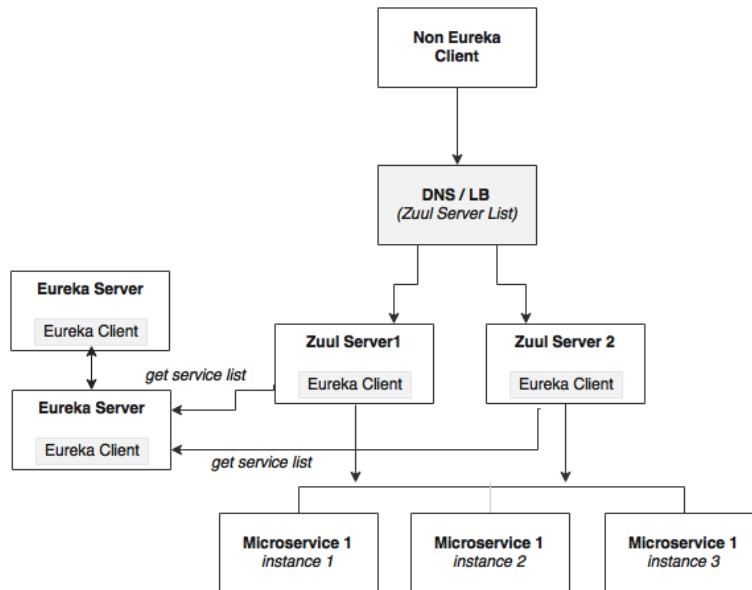
As shown in the preceding diagram, Zuul services register themselves with Eureka with a service ID, `search-apigateway` in our case. The Eureka client asks for the server list with the ID `search-apigateway`. The Eureka server returns the list of servers based on the current Zuul topology. The Eureka client, based on this list picks up one of the servers, and initiates the call.

As we saw earlier, the client uses the service ID to resolve the Zuul instance. In the following case, `search-apigateway` is the Zuul instance ID registered with Eureka:

```
Flight[] flights = searchClient.postForObject("http://search-
apigateway/api/search/get", searchQuery, Flight[].class);
```

## High availability when the client is not a Eureka client

In this case, the client is not capable of handling load balancing by using the Eureka server. As shown in the following diagram, the client sends the request to a load balancer, which in turn identifies the right Zuul service instance. The Zuul instance, in this case, will be running behind a load balancer such as HAProxy or a hardware load balancer like NetScaler:



The microservices will still be load balanced by Zuul using the Eureka server.

## Completing Zuul for all other services

In order to complete this exercise, add API gateway projects (name them as `*-apigateway`) for all our microservices. The following steps are required to achieve this task:

1. Create new property files per service, and check in to the Git repositories
2. Change `application.properties` to `bootstrap.properties`, and add the required configurations
3. Add `@EnableZuulProxy` to `Application.java` in each of the `*-apigateway` projects.

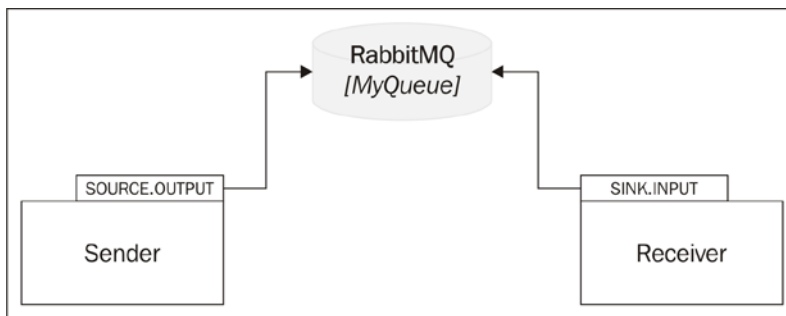
4. Add `@EnableDiscoveryClient` in all the `Application.java` files under each of the `*-apigateway` projects.
5. Optionally, change the package names and file names generated by default

In the end, we will have the following API gateway projects:

- `chapter5.fares-apigateway`
- `chapter5.search-apigateway`
- `chapter5.checkin-apigateway`
- `chapter5.book-apigateway`

## Streams for reactive microservices

Spring Cloud Stream provides an abstraction over the messaging infrastructure. The underlying messaging implementation can be RabbitMQ, Redis, or Kafka. Spring Cloud Stream provides a declarative approach for sending and receiving messages:

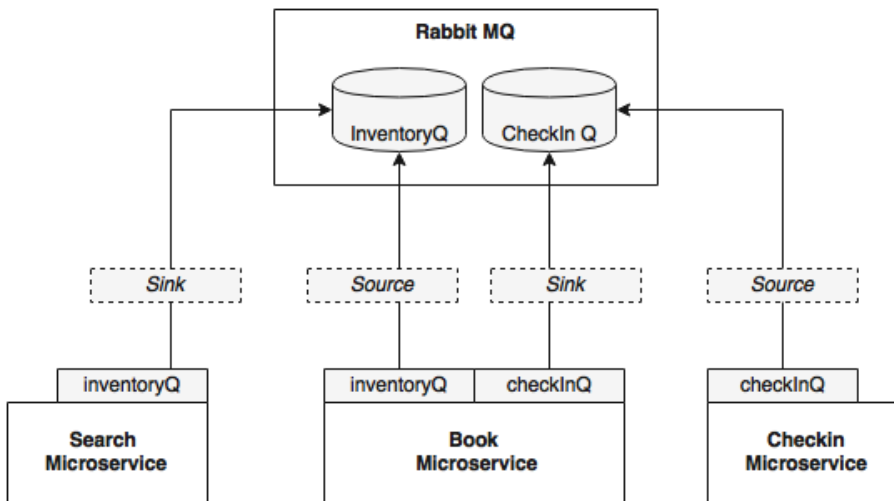


As shown in the preceding diagram, Cloud Stream works on the concept of a **source** and a **sink**. The source represents the sender perspective of the messaging, and sink represents the receiver perspective of the messaging.

In the example shown in the diagram, the sender defines a logical queue called `Source.OUTPUT` to which the sender sends messages. The receiver defines a logical queue called `Sink.INPUT` from which the receiver retrieves messages. The physical binding of `OUTPUT` to `INPUT` is managed through the configuration. In this case, both link to the same physical queue—`MyQueue` on RabbitMQ. So, while at one end, `Source.OUTPUT` points to `MyQueue`, on the other end, `Sink.INPUT` points to the same `MyQueue`.

Spring Cloud offers the flexibility to use multiple messaging providers in one application such as connecting an input stream from Kafka to a Redis output stream, without managing the complexities. Spring Cloud Stream is the basis for message-based integration. The Cloud Stream Modules subproject is another Spring Cloud library that provides many endpoint implementations.

As the next step, rebuild the inter-microservice messaging communication with the Cloud Streams. As shown in the next diagram, we will define a `SearchSink` connected to `InventoryQ` under the `Search` microservice. `Booking` will define a `BookingSource` for sending inventory change messages connected to `InventoryQ`. Similarly, `Check-in` defines a `CheckinSource` for sending the check-in messages. `Booking` defines a sink, `BookingSink`, for receiving messages, both bound to the `CheckinQ` queue on the `RabbitMQ`:



In this example, we will use RabbitMQ as the message broker:

1. Add the following Maven dependency to `Booking`, `Search`, and `Check-in`, as these are the three modules using messaging:

```

<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-stream-rabbit
 </artifactId>
</dependency>

```

2. Add the following two properties to `booking-service.properties`. These properties bind the logical queue `inventoryQ` to physical `inventoryQ`, and the logical `checkinQ` to the physical `checkinQ`:

```
spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

3. Add the following property to `search-service.properties`. This property binds the logical queue `inventoryQ` to the physical `inventoryQ`:

```
spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ
```

4. Add the following property to `checkin-service.properties`. This property binds the logical queue `checkinQ` to the physical `checkinQ`:

```
spring.cloud.stream.bindings.checkInQ.destination=checkInQ
```

5. Commit all files to the Git repository
6. The next step is to edit the code. The Search microservice consumes a message from the Booking microservice. In this case, Booking is the source and Search is the sink.

Add `@EnableBinding` to the Sender class of the Booking service. This enables the Cloud Stream to work on autoconfigurations based on the message broker library available in the class path. In our case, it is `RabbitMQ`. The parameter `BookingSource` defines the logical channels to be used for this configuration:

```
@EnableBinding(BookingSource.class)
public class Sender {
```

7. In this case, `BookingSource` defines a message channel called `inventoryQ`, which is physically bound to `RabbitMQ`'s `inventoryQ`, as configured in the configuration. `BookingSource` uses an annotation, `@Output`, to indicate that this is of the output type—a message that is outgoing from a module. This information will be used for autoconfiguration of the message channel

```
interface BookingSource {
 public static String InventoryQ="inventoryQ";
 @Output("inventoryQ")
 public MessageChannel inventoryQ();
}
```

8. Instead of defining a custom class, we can also use the default `Source` class that comes with Spring Cloud Stream if the service has only one source and sink:

```
public interface Source {
 @Output("output")
```

```

 MessageChannel output();
 }

```

9. Define a message channel in the sender, based on `BookingSource`. The following code will inject an output message channel with the name `inventory`, which is already configured in `BookingSource`:

```

 @Output (BookingSource.InventoryQ)
 @Autowired
 private MessageChannel;

```

10. Reimplement the `send` message method in `BookingSender`:

```

public void send(Object message) {
 messageChannel.
 send(MessageBuilder.withPayload(message).
 build());
}

```

11. Now add the following to the `SearchReceiver` class the same way we did for the `Booking` service:

```

@EnableBinding(SearchSink.class)
public class Receiver {

```

12. In this case, the `SearchSink` interface will look like the following. This will define the logical sink queue it is connected with. The message channel in this case is defined as `@Input` to indicate that this message channel is to accept messages:

```

interface SearchSink {
 public static String INVENTORYQ="inventoryQ";
 @Input("inventoryQ")
 public MessageChannel inventoryQ();
}

```

13. Amend the `Search` service to accept this message:

```

@ServiceActivator(inputChannel = SearchSink.INVENTORYQ)
public void accept(Map<String, Object> fare) {
 searchComponent.updateInventory((String) fare.
 get("FLIGHT_NUMBER"), (String) fare.
 get("FLIGHT_DATE"), (int) fare.
 get("NEW_INVENTORY"));
}

```



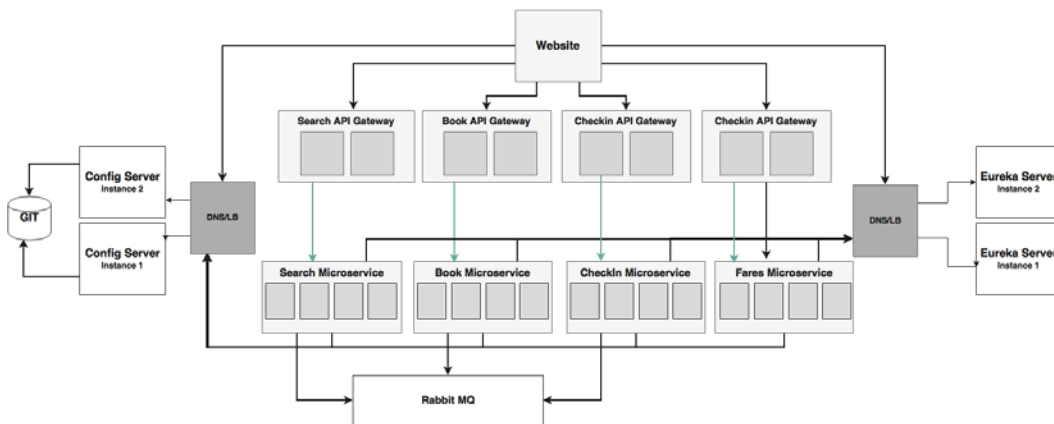
14. We will still need the RabbitMQ configurations that we have in our configuration files to connect to the message broke

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
server.port=8090
```

15. Run all services, and run the website project. If everything is fine, the website project successfully executes the Search, Booking, and Check-in functions. The same can also be tested using the browser by pointing to `http://localhost:8001`.

## Summarizing the BrownField PSS architecture

The following diagram shows the overall architecture that we have created with the Config server, Eureka, Feign, Zuul, and Cloud Streams. The architecture also includes the high availability of all components. In this case, we assume that the client uses the Eureka client libraries:



The summary of the projects and the port they are listening on is given in the following table:

Microservice	Projects	Port
Book microservice	chapter5.book	8060 to 8064
Check-in microservice	chapter5.checkin	8070 to 8074
Fare microservice	chapter5.fares	8080 to 8084
Search microservice	chapter5.search	8090 to 8094
Website client	chapter5.website	8001
Spring Cloud Config server	chapter5.configserver	8888/8889
Spring Cloud Eureka server	chapter5.eurekaserver	8761/8762
Book API gateway	chapter5.book-apigateway	8095 to 8099
Check-in API gateway	chapter5.checkin-apigateway	8075 to 8079
Fares API gateway	chapter5.fares-apigateway	8085 to 8089
Search API gateway	chapter5.search-apigateway	8065 to 8069

Follow these steps to do a final run

1. Run RabbitMQ.
2. Build all projects using `pom.xml` at the root level:  

```
mvn -Dmaven.test.skip=true clean install
```
3. Run the following projects from their respective folders. Remember to wait for 40 to 50 seconds before starting the next service. This will ensure that the dependent services are registered and are available before we start a new service:

```
java -jar target/fares-1.0.jar
java -jar target/search-1.0.jar
java -jar target/checkin-1.0.jar
java -jar target/book-1.0.jar
java -jar target/fares-apigateway-1.0.jar
java -jar target/search-apigateway-1.0.jar
java -jar target/checkin-apigateway-1.0.jar
java -jar target/book-apigateway-1.0.jar
java -jar target/website-1.0.jar
```

4. Open the browser window, and point to `http://localhost:8001`. Follow the steps mentioned in the *Running and testing the project* section in *Chapter 4, Microservices Evolution – A Case Study*.

## Summary

In this chapter, you learned how to scale a Twelve-Factor Spring Boot microservice using the Spring Cloud project. What you learned was then applied to the BrownField Airline's PSS microservice that we developed in the previous chapter.

We then explored the Spring Config server for externalizing the microservices' configuration, and the way to deploy the Config server for high availability. We also discussed the declarative service calls using Feign, examined the use of Ribbon and Eureka for load balancing, dynamic service registration, and discovery. Implementation of an API gateway was examined by implementing Zuul. Finally, we concluded with a reactive style integration of microservices using Spring Cloud Stream.

BrownField Airline's PSS microservices are now deployable on the Internet scale. Other Spring Cloud components such as Hyterix, Sleuth, and so on will be covered in *Chapter 7, Logging and Monitoring Microservices*. The next chapter will demonstrate autoscaling features, extending the BrownField PSS implementation.

# 6

## Autoscaling Microservices

Spring Cloud provides the support essential for the deployment of microservices at scale. In order to get the full power of a cloud-like environment, the microservices instances should also be capable of scaling out and shrinking automatically based on traffic patterns

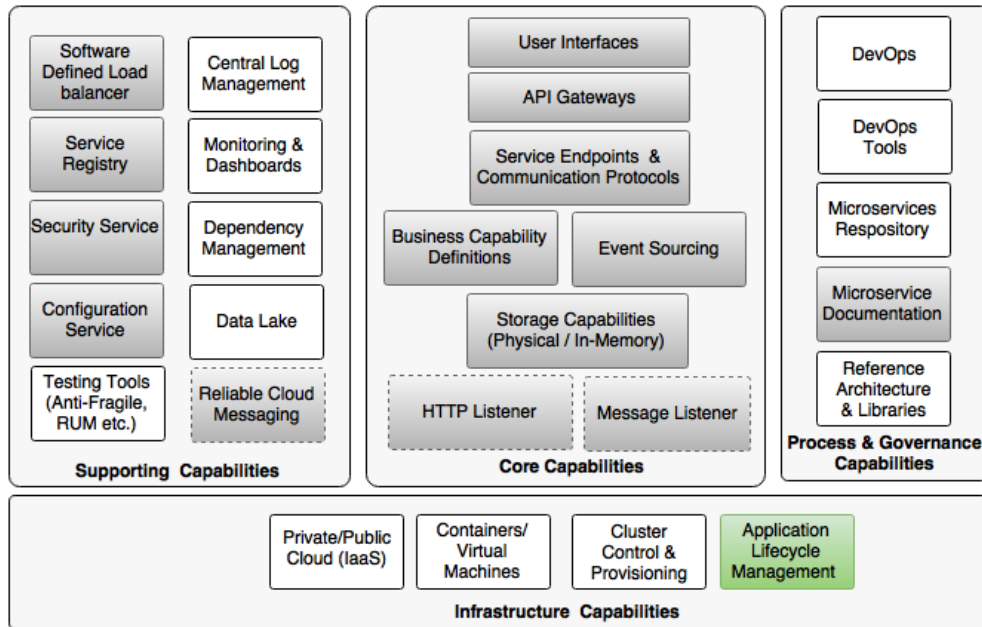
This chapter will detail out how to make microservices elastically grow and shrink by effectively using the actuator data collected from Spring Boot microservices to control the deployment topology by implementing a simple life cycle manager.

By the end of this chapter, you will learn about the following topics:

- The basic concept of autoscaling and different approaches for autoscaling
- The importance and capabilities of a life cycle manager in the context of microservices
- Examining the custom life cycle manager to achieve autoscaling
- Programmatically collecting statistics from the Spring Boot actuator and using it to control and shape incoming traffic

# Reviewing the microservice capability model

This chapter will cover the **Application Lifecycle Management** capability in the microservices capability model discussed in *Chapter 3, Applying Microservices Concepts*, highlighted in the following diagram:



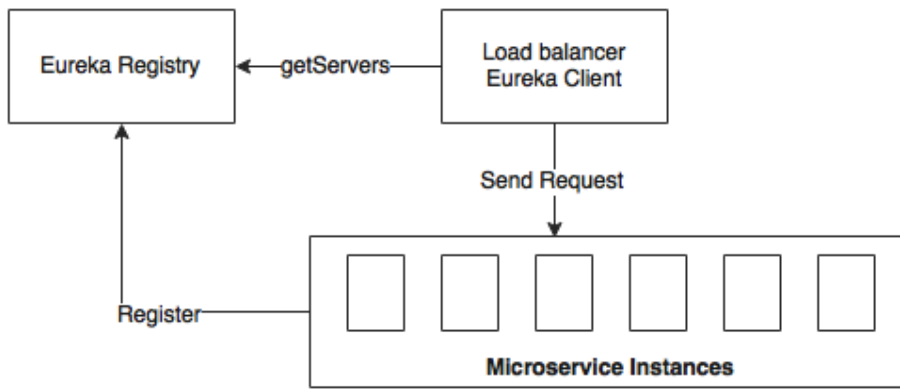
We will see a basic version of the life cycle manager in this chapter, which will be enhanced in later chapters.

## Scaling microservices with Spring Cloud

In *Chapter 5, Scaling Microservices with Spring Cloud*, you learned how to scale Spring Boot microservices using Spring Cloud components. The two key concepts of Spring Cloud that we implemented are self-registration and self-discovery. These two capabilities enable automated microservices deployments. With self-registration, microservices can automatically advertise the service availability by registering service metadata to a central service registry as soon as the instances are ready to accept traffic. Once the microservices are registered, consumers can consume the newly registered services from the very next moment by discovering service instances using the registry service. Registry is at the heart of this automation.

This is quite different from the traditional clustering approach employed by the traditional JEE application servers. In the case of JEE application servers, the server instances' IP addresses are more or less statically configured in a load balancer. Therefore, the cluster approach is not the best solution for automatic scaling in Internet-scale deployments. Also, clusters impose other challenges, such as they have to have exactly the same version of binaries on all cluster nodes. It is also possible that the failure of one cluster node can poison other nodes due to the tight dependency between nodes.

The registry approach decouples the service instances. It also eliminates the need to manually maintain service addresses in the load balancer or configure virtual IPs



As shown in the diagram, there are three key components in our automated microservices deployment topology:

- **Eureka** is the central registry component for microservice registration and discovery. REST APIs are used by both consumers as well as providers to access the registry. The registry also holds the service metadata such as the service identity, host, port, health status, and so on.
- The **Eureka** client, together with the **Ribbon** client, provide client-side dynamic load balancing. Consumers use the Eureka client to look up the Eureka server to identify the available instances of a target service. The Ribbon client uses this server list to load-balance between the available microservice instances. In a similar way, if the service instance goes out of service, these instances will be taken out of the Eureka registry. The load balancer automatically reacts to these dynamic topology changes.
- The third component is the **microservices** instances developed using Spring Boot with the actuator endpoints enabled.

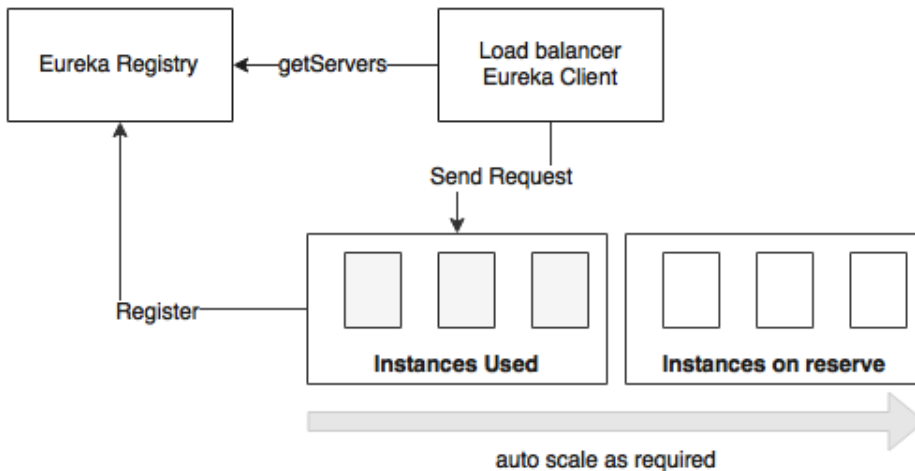
However, there is one gap in this approach. When there is need for an additional microservice instance, a manual task is required to kick off a new instance. In an ideal scenario, the starting and stopping of microservice instances also require automation.

For example, when there is a requirement to add another Search microservice instance to handle the increase in traffic volumes or a load burst scenario, the administrator has to manually bring up a new instance. Also, when the Search instance is idle for some time, it needs to be manually taken out of service to have optimal infrastructure usage. This is especially relevant when services run on a pay-as-per-usage cloud environment.

## Understanding the concept of autoscaling

Autoscaling is an approach to automatically scaling out instances based on the resource usage to meet the SLAs by replicating the services to be scaled.

The system automatically detects an increase in traffic, spins up additional instances, and makes them available for traffic handling. Similarly, when the traffic volume goes down, the system automatically detects and reduces the number of instances by taking active instances back from the service:



As shown in the preceding diagram, autoscaling is done, generally, using a set of reserve machines.

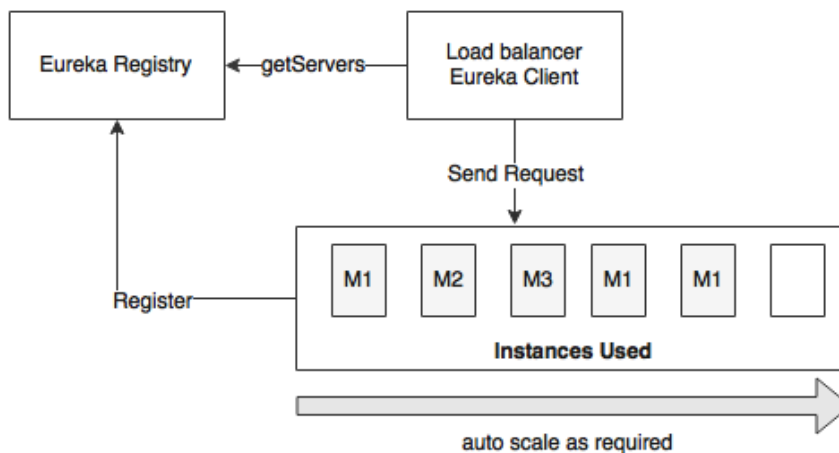
As many of the cloud subscriptions are based on a pay-as-you-go model, this is an essential capability when targeting cloud deployments. This approach is often called **elasticity**. It is also called **dynamic resource provisioning and deprovisioning**. Autoscaling is an effective approach specifically for microservices with varying traffic patterns. For example, an Accounting service would have high traffic during month ends and year ends. There is no point in permanently provisioning instances to handle these seasonal loads.

In the autoscaling approach, there is often a resource pool with a number of spare instances. Based on the demand, instances will be moved from the resource pool to the active state to meet the surplus demand. These instances are not pretagged for any particular microservices or prepackaged with any of the microservice binaries. In advanced deployments, the Spring Boot binaries are downloaded on demand from an artifact repository such as Nexus or Artifactory.

## The benefits of autoscaling

There are many benefits in implementing the autoscaling mechanism. In traditional deployments, administrators reserve a set of servers against each application. With autoscaling, this preallocation is no longer required. This prefixed server allocation may result in underutilized servers. In this case, idle servers cannot be utilized even when neighboring services struggle for additional resources.

With hundreds of microservice instances, preallocating a fixed number of servers to each of the microservices is not cost effective. A better approach is to reserve a number of server instances for a group of microservices without preallocating or tagging them against a microservice. Instead, based on the demand, a group of services can share a set of available resources. By doing so, microservices can be dynamically moved across the available server instances by optimally using the resources:





As shown in the preceding diagram, there are three instances of the **M1** microservice, one instance of **M2**, and one instance of **M3** up and running. There is another server kept unallocated. Based on the demand, the unallocated server can be used for any of the microservices: **M1**, **M2**, or **M3**. If **M1** has more service requests, then the unallocated instance will be used for **M1**. When the service usage goes down, the server instance will be freed up and moved back to the pool. Later, if the **M2** demand increases, the same server instance can be activated using **M2**.

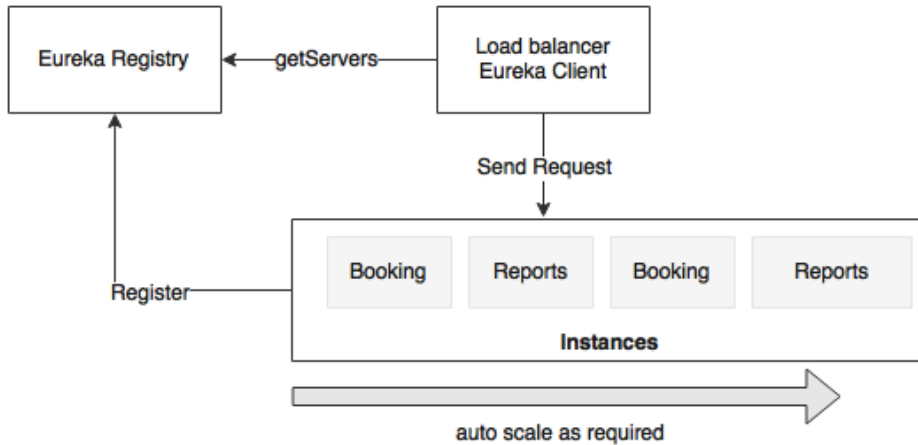
Some of the key benefits of autoscaling are

- **It has high availability and is fault tolerant:** As there are multiple service instances, even if one fails, another instance can take over and continue serving clients. This failover will be transparent to the consumers. If no other instance of this service is available, the autoscaling service will recognize this situation and bring up another server with the service instance. As the whole process of bringing up or bringing down instances is automatic, the overall availability of the services will be higher than the systems implemented without autoscaling. The systems without autoscaling require manual intervention to add or remove service instances, which will be hard to manage in large deployments.

For example, assume that two of instances of the Booking service are running. If there is an increase in the traffic flow, in a normal scenario, the existing instance might become overloaded. In most of the scenarios, the entire set of services will be jammed, resulting in service unavailability. In the case of autoscaling, a new Booking service instance can be brought up quickly. This will balance the load and ensure service availability.

- **It increases scalability:** One of the key benefits of autoscaling is horizontal scalability. Autoscaling allows us to selectively scale up or scale down services automatically based on traffic patterns.
- **It has optimal usage and is cost saving:** In a pay-as-you-go subscription model, billing is based on actual resource utilization. With the autoscaling approach, instances will be started and shut down based on the demand. Hence, resources are optimally utilized, thereby saving cost.

- **It gives priority to certain services or group of services:** With autoscaling, it is possible to give priority to certain critical transactions over low-value transactions. This will be done by removing an instance from a low-value service and reallocating it to a high-value service. This will also eliminate situations where a low-priority transaction heavily utilizes resources when high-value transactions are cramped up for resources.



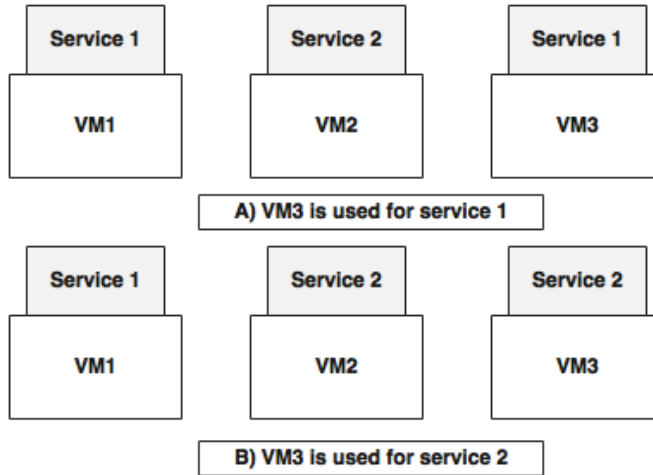
For instance, the **Booking** and **Reports** services run with two instances, as shown in the preceding diagram. Let's assume that the **Booking** service is a revenue generation service and therefore has a higher value than the **Reports** service. If there are more demands for the **Booking** service, then one can set policies to take one **Reports** service out of the service and release this server for the **Booking** service.

## Different autoscaling models

Autoscaling can be applied at the application level or at the infrastructure level. In a nutshell, application scaling is scaling by replicating application binaries only, whereas infrastructure scaling is replicating the entire virtual machine, including application binaries.

## Autoscaling an application

In this scenario, scaling is done by replicating the microservices, not the underlying infrastructure, such as virtual machines. The assumption is that there is a pool of VMs or physical infrastructures available to scale up microservices. These VMs have the basic image fused with any dependencies, such as JRE. It is also assumed that microservices are homogeneous in nature. This gives flexibility in reusing the same virtual or physical machines for different services:

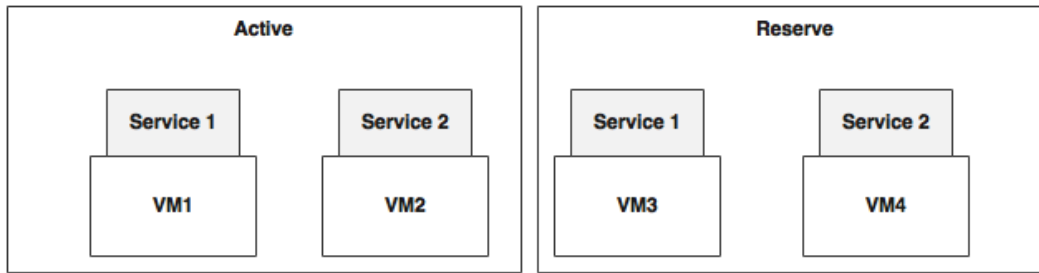


As shown in the preceding diagram, in scenario **A**, **VM3** is used for **Service 1**, whereas in scenario **B**, the same **VM3** is used for **Service 2**. In this case, we only swapped the application library and not the underlying infrastructure.

This approach gives faster instantiation as we are only handling the application binaries and not the underlying VMs. The switching is easier and faster as the binaries are smaller in size and there is no OS boot required either. However, the downside of this approach is that if certain microservices require OS-level tuning or use polyglot technologies, then dynamically swapping microservices will not be effective.

## Autoscaling the infrastructure

In contrast to the previous approach, in this case, the infrastructure is also provisioned automatically. In most cases, this will create a new VM on the fly or destroy the VM based on the demand:



As shown in the preceding diagram, the reserve instances are created as VM images with predefined service instances. When there is demand for **Service 1**, **VM3** is moved to an active state. When there is a demand for **Service 2**, **VM4** is moved to the active state.

This approach is efficient if the applications depend upon the parameters and libraries at the infrastructure level, such as the operating system. Also, this approach is better for polyglot microservices. The downside is the heavy nature of VM images and the time required to spin up a new VM. Lightweight containers such as Docker are preferred in such cases instead of traditional heavyweight virtual machines.

## Autoscaling in the cloud

Elasticity or autoscaling is one of the fundamental features of most cloud providers. Cloud providers use infrastructure scaling patterns, as discussed in the previous section. These are typically based on a set of pooled machines.

For example, in AWS, these are based on introducing new EC2 instances with a predefined AMI. AWS supports autoscaling with the help of autoscaling groups. Each group is set with a minimum and maximum number of instances. AWS ensures that the instances are scaled on demand within these bounds. In case of predictable traffic patterns, provisioning can be configured based on timelines. AWS also provides ability for applications to customize autoscaling policies.

Microsoft Azure also supports autoscaling based on the utilization of resources such as the CPU, message queue length, and so on. IBM Bluemix supports autoscaling based on resources such as CPU usage.

Other PaaS platforms, such as CloudBees and OpenShift, also support autoscaling for Java applications. Pivotal Cloud Foundry supports autoscaling with the help of Pivotal Autoscale. Scaling policies are generally based on resource utilization, such as the CPU and memory thresholds.

There are components that run on top of the cloud and provide fine-grained controls to handle autoscaling. Netflix Fenzo, Eucalyptus, Boxfuse, and Mesosphere are some of the components in this category.

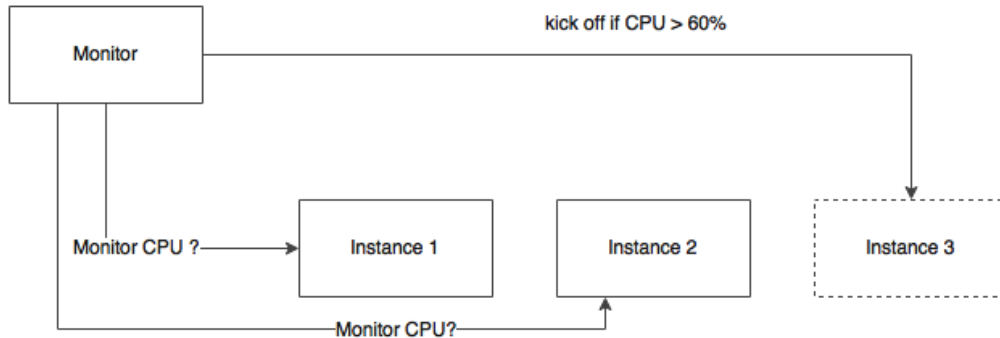
## Autoscaling approaches

Autoscaling is handled by considering different parameters and thresholds. In this section, we will discuss the different approaches and policies that are typically applied to take decisions on when to scale up or down.

### Scaling with resource constraints

This approach is based on real-time service metrics collected through monitoring mechanisms. Generally, the resource-scaling approach takes decisions based on the CPU, memory, or the disk of machines. This can also be done by looking at the statistics collected on the service instances themselves, such as heap memory usage.

A typical policy may be spinning up another instance when the CPU utilization of the machine goes beyond 60%. Similarly, if the heap size goes beyond a certain threshold, we can add a new instance. The same applies to downsizing the compute capacity when the resource utilization goes below a set threshold. This is done by gradually shutting down servers:



In typical production scenarios, the creation of additional services is not done on the first occurrence of a threshold breach. The most appropriate approach is to define sliding window or a waiting period.

The following are some of the examples:

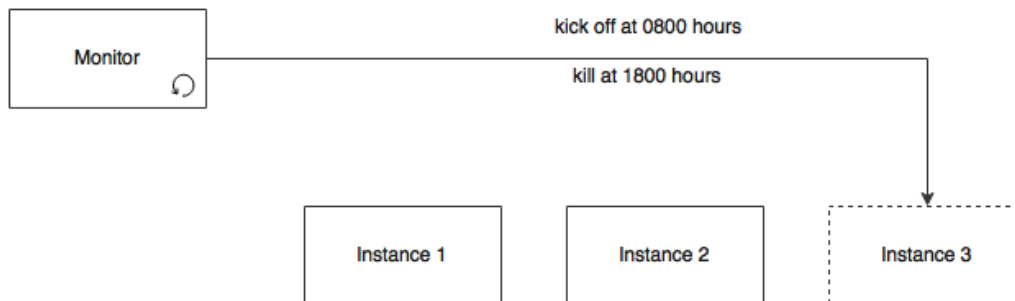
- An example of a **response sliding window** is if 60% of the response time of a particular transaction is consistently more than the set threshold value in a 60-second sampling window, increase service instances
- In a **CPU sliding window**, if the CPU utilization is consistently beyond 70% in a 5 minutes sliding window, then a new instance is created
- An example of the **exception sliding window** is if 80% of the transactions in a sliding window of 60 seconds or 10 consecutive executions result in a particular system exception, such as a connection timeout due to exhausting the thread pool, then a new service instance is created

In many cases, we will set a lower threshold than the actual expected thresholds. For example, instead of setting the CPU utilization threshold at 80%, set it at 60% so that the system gets enough time to spin up an instance before it stops responding. Similarly, when scaling down, we use a lower threshold than the actual. For example, we will use 40% CPU utilization to scale down instead of 60%. This allows us to have a cool-down period so that there will not be any resource struggle when shutting down instances.

Resource-based scaling is also applicable to service-level parameters such as the throughput of the service, latency, applications thread pool, connection pool, and so on. These can also be at the application level, such as the number of **sales orders** processing in a service instance, based on internal benchmarking.

## Scaling during specific time periods

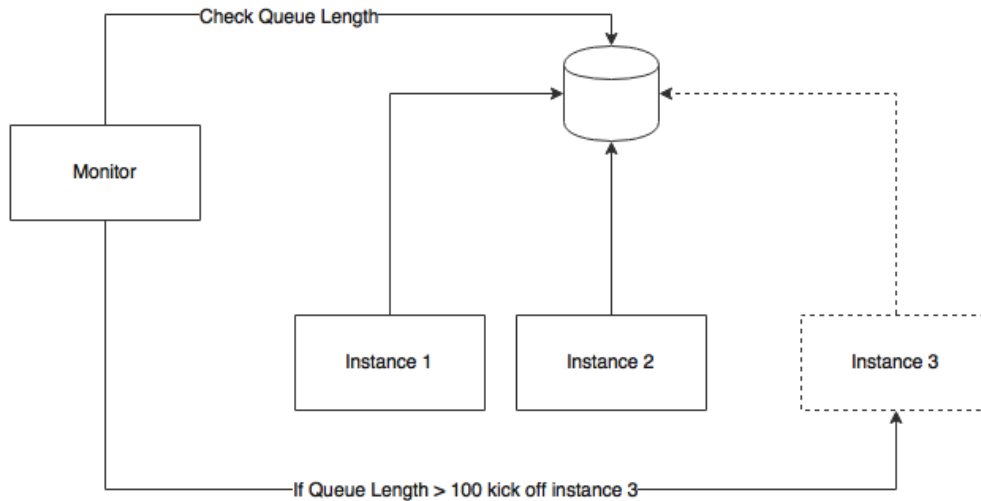
Time-based scaling is an approach to scaling services based on certain periods of the day, month, or year to handle seasonal or business peaks. For example, some services may experience a higher number of transactions during office hours and a considerably low number of transactions outside office hours. In this case, during the day, services autoscale to meet the demand and automatically downsize during the non-office hours



Many airports worldwide impose restrictions on night-time landing. As a result, the number of passengers checking in at the airports during the night time is less compared to the day time. Hence, it is cost effective to reduce the number of instances during the night time.

## Scaling based on the message queue length

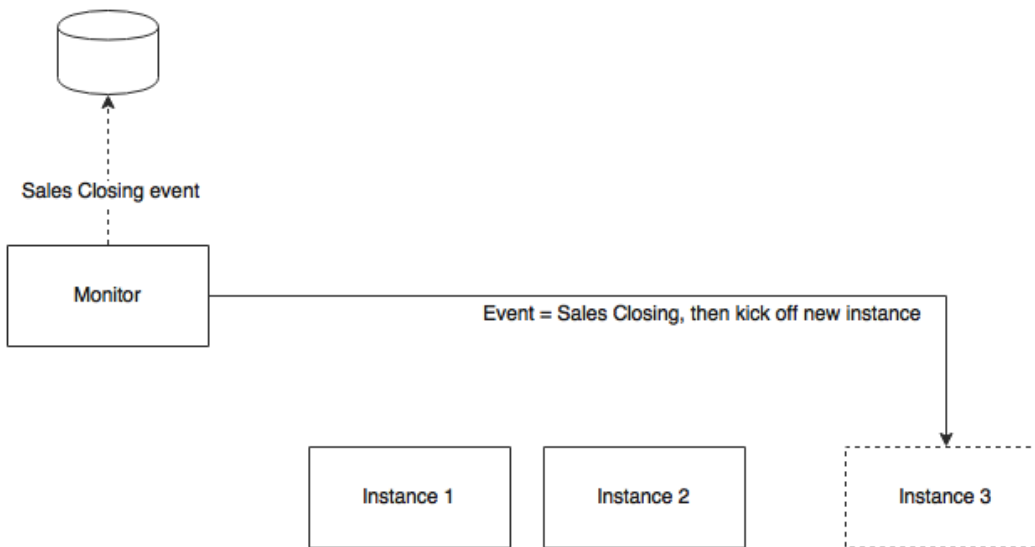
This is particularly useful when the microservices are based on asynchronous messaging. In this approach, new consumers are automatically added when the messages in the queue go beyond certain limits:



This approach is based on the competing consumer pattern. In this case, a pool of instances is used to consume messages. Based on the message threshold, new instances are added to consume additional messages.

## Scaling based on business parameters

In this case, adding instances is based on certain business parameters—for example, spinning up a new instance just before handling **sales closing** transactions. As soon as the monitoring service receives a preconfigured business event (such as **sales closing minus 1 hour**), a new instance will be brought up in anticipation of large volumes of transactions. This will provide fine-grained control on scaling based on business rules:



## Predictive autoscaling

Predictive scaling is a new paradigm of autoscaling that is different from the traditional real-time metrics-based autoscaling. A prediction engine will take multiple inputs, such as historical information, current trends, and so on, to predict possible traffic patterns. Autoscaling is done based on these predictions. Predictive autoscaling helps avoid hardcoded rules and time windows. Instead, the system can automatically predict such time windows. In more sophisticated deployments, predictive analysis may use cognitive computing mechanisms to predict autoscaling.

In the cases of sudden traffic spikes, traditional autoscaling may not help. Before the autoscaling component can react to the situation, the spike would have hit and damaged the system. The predictive system can understand these scenarios and predict them before their actual occurrence. An example will be handling a flood of requests immediately after a planned outage.

Netflix Scryster is an example of such a system that can predict resource requirements in advance.



# Autoscaling BrownField PSS microservices

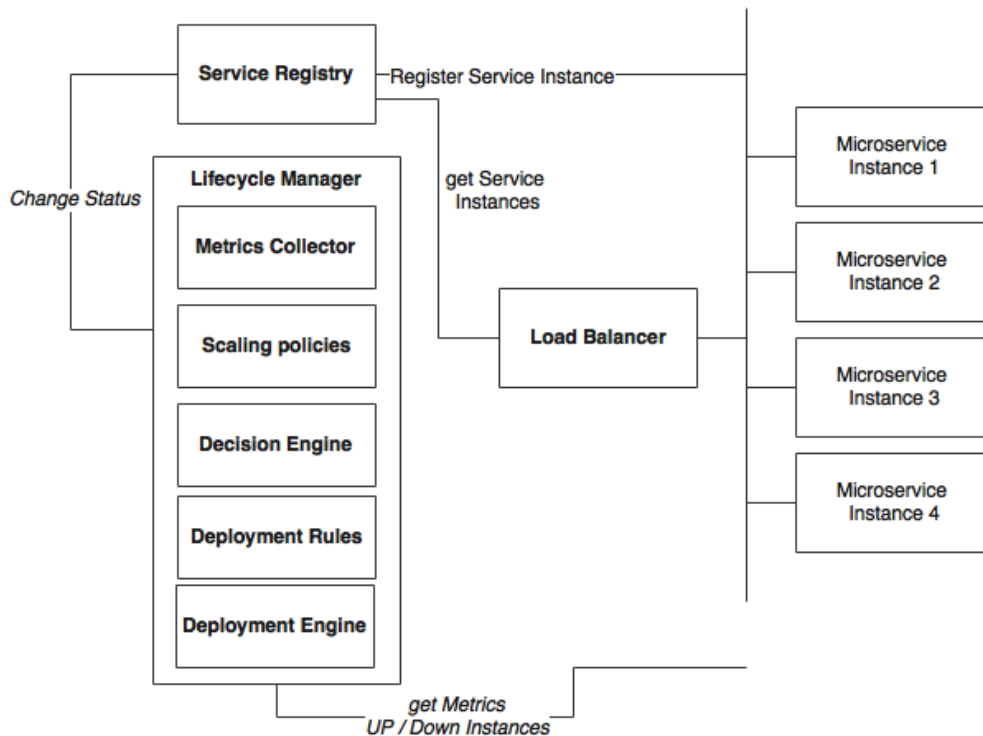
In this section, we will examine how to enhance microservices developed in *Chapter 5, Scaling Microservices with Spring Cloud*, for autoscaling. We need a component to monitor certain performance metrics and trigger autoscaling. We will call this component the **life cycle manager**.

The service life cycle manager, or the application life cycle manager, is responsible for detecting scaling requirements and adjusting the number of instances accordingly. It is responsible for starting and shutting down instances dynamically.

In this section, we will take a look at a primitive autoscaling system to understand the basic concepts, which will be enhanced in later chapters.

## The capabilities required for an autoscaling system

A typical autoscaling system has capabilities as shown in the following diagram:



The components involved in the autoscaling ecosystem in the context of microservices are explained as follows:

- **Microservices:** These are sets of the up-and-running microservice instances that keep sending health and metrics information. Alternately, these services expose actuator endpoints for metrics collection. In the preceding diagram, these are represented as **Microservice 1** through **Microservice 4**.
- **Service Registry:** A service registry keeps track of all the services, their health states, their metadata, and their endpoint URI.
- **Load Balancer:** This is a client-side load balancer that looks up the service registry to get up-to-date information about the available service instances.
- **Lifecycle Manager:** The life cycle manager is responsible for autoscaling, which has the following subcomponents:
  - **Metrics Collector:** A metrics collection unit is responsible for collecting metrics from all service instances. The life cycle manager will aggregate the metrics. It may also keep a sliding time window. The metrics could be infrastructure-level metrics, such as CPU usage, or application-level metrics, such as transactions per minute.
  - **Scaling policies:** Scaling policies are nothing but sets of rules indicating when to scale up and scale down microservices—for example, 90% of CPU usage above 60% in a sliding window of 5 minutes.
  - **Decision Engine:** A decision engine is responsible for making decisions to scale up and scale down based on the aggregated metrics and scaling policies.
  - **Deployment Rules:** The deployment engine uses deployment rules to decide which parameters to consider when deploying services. For example, a service deployment constraint may say that the instance must be distributed across multiple availability regions or a 4 GB minimum of memory required for the service.
  - **Deployment Engine:** The deployment engine, based on the decisions of the decision engine, can start or stop microservice instances or update the registry by altering the health states of services. For example, it sets the health status as "out of service" to take out a service temporarily.

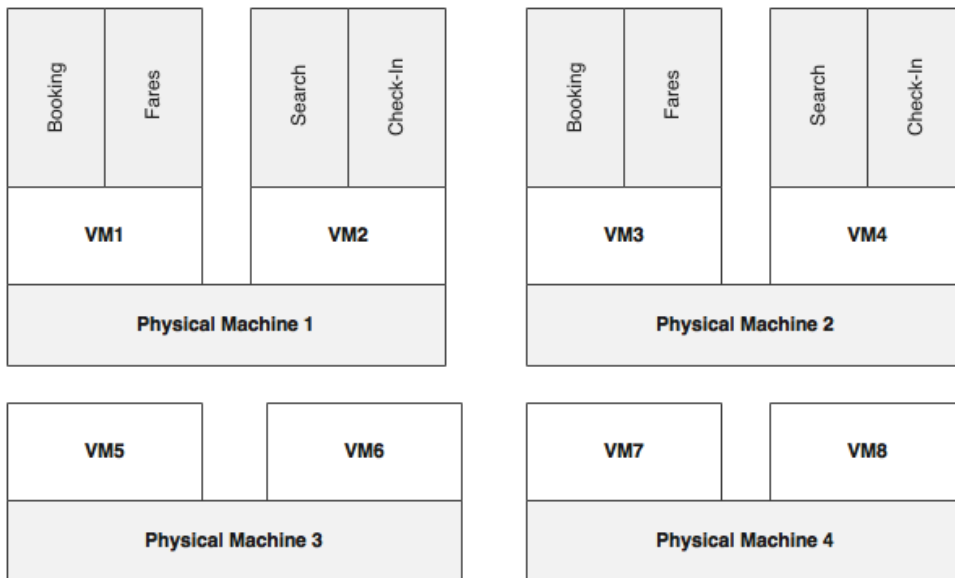
## Implementing a custom life cycle manager using Spring Boot

The life cycle manager introduced in this section is a minimal implementation to understand autoscaling capabilities. In later chapters, we will enhance this implementation with containers and cluster management solutions. Ansible, Marathon, and Kubernetes are some of the tools useful in building this capability.

In this section, we will implement an application-level autoscaling component using Spring Boot for the services developed in *Chapter 5, Scaling Microservices with Spring Cloud*.

## Understanding the deployment topology

The following diagram shows a sample deployment topology of BrownField PSS microservices:



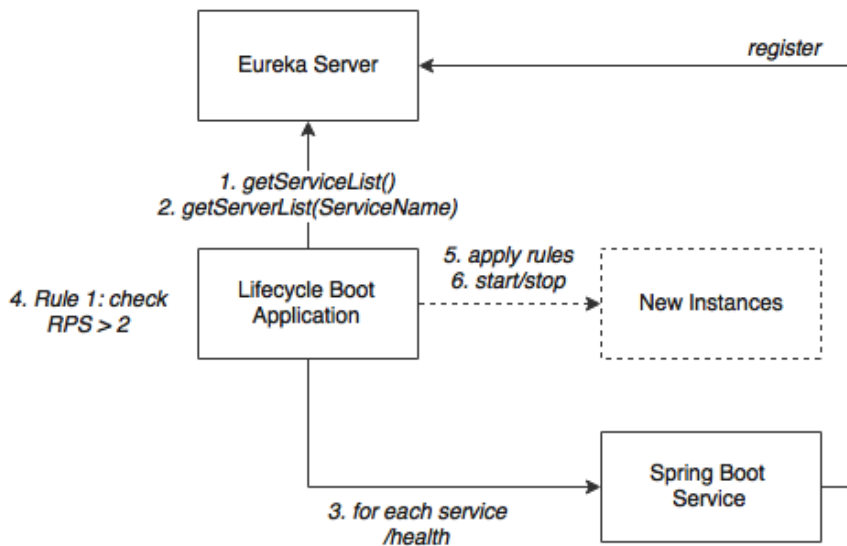
As shown in the diagram, there are four physical machines. Eight VMs are created from four physical machines. Each physical machine is capable of hosting two VMs, and each VM is capable of running two Spring Boot instances, assuming that all services have the same resource requirements.

Four VMs, **VM1** through **VM4**, are active and are used to handle traffic. **VM5** to **VM8** are kept as reserve VMs to handle scalability. **VM5** and **VM6** can be used for any of the microservices and can also be switched between microservices based on scaling demands. Redundant services use VMs created from different physical machines to improve fault tolerance.

Our objective is to scale out any services when there is increase in traffic flow using four VMs, **VM5** through **VM8**, and scale down when there is not enough load. The architecture of our solution is as follows.

## Understanding the execution flow

Have a look at the following flowchart



As shown in the preceding diagram, the following activities are important for us:

- The Spring Boot service represents microservices such as Search, Book, Fares, and Check-in. Services at startup automatically register endpoint details to the Eureka registry. These services are actuator-enabled, so the life cycle manager can collect metrics from the actuator endpoints.

- The life cycle manager service is nothing but another Spring Boot application. The life cycle manager has a metrics collector that runs a background job, periodically polls the Eureka server, and gets details of all the service instances. The metrics collector then invokes the actuator endpoints of each microservice registered in the Eureka registry to get the health and metrics information. In a real production scenario, a subscription approach for data collection is better.
- With the collected metrics information, the life cycle manager executes a list of policies and derives decisions on whether to scale up or scale down instances. These decisions are either to start a new service instance of a particular type on a particular VM or to shut down a particular instance.
- In the case of shutdown, it connects to the server using an actuator endpoint and calls the shutdown service to gracefully shut down an instance.
- In the case of starting a new instance, the deployment engine of the life cycle manager uses the scaling rules and decides where to start the new instance and what parameters are to be used when starting the instance. Then, it connects to the respective VMs using SSH. Once connected, it executes a preinstalled script (or passes this script as a part of the execution) by passing the required constraints as a parameter. This script fetches the application library from a central Nexus repository in which the production binaries are kept and initiates it as a Spring Boot application. The port number is parameterized by the life cycle manager. SSH needs to be enabled on the target machines.

In this example, we will use **TPM (Transactions Per Minute)** or **RPM (Requests Per Minute)** as sampler metrics for decision making. If the Search service has more than 10 TPM, then it will spin up a new Search service instance. Similarly, if the TPM is below 2, one of the instances will be shut down and released back to the pool.

When starting a new instance, the following policies will be applied:

- The number of service instances at any point should be a minimum of 1 and a maximum of 4. This also means that at least one service instance will always be up and running.
- A scaling group is defined in such a way that a new instance is created on a VM that is on a different physical machine. This will ensure that the services run across different physical machines.

These policies could be further enhanced. The life cycle manager ideally provides options to customize these rules through REST APIs or Groovy scripts.

# A walkthrough of the life cycle manager code

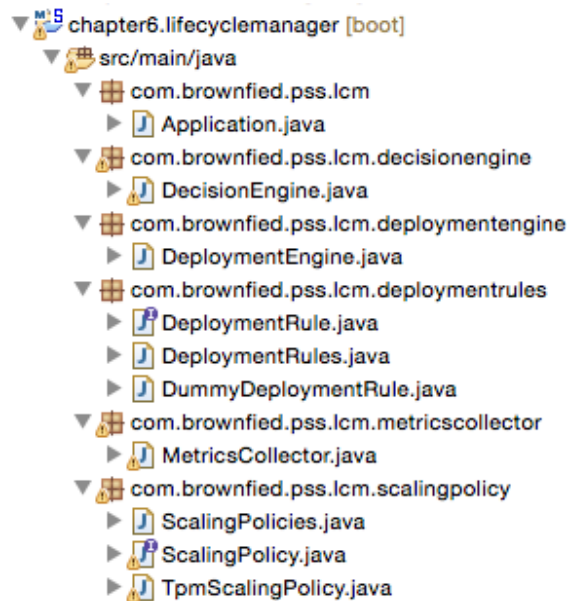
We will take a look at how a simple life cycle manager is implemented. This section will be a walkthrough of the code to understand the different components of the life cycle manager.



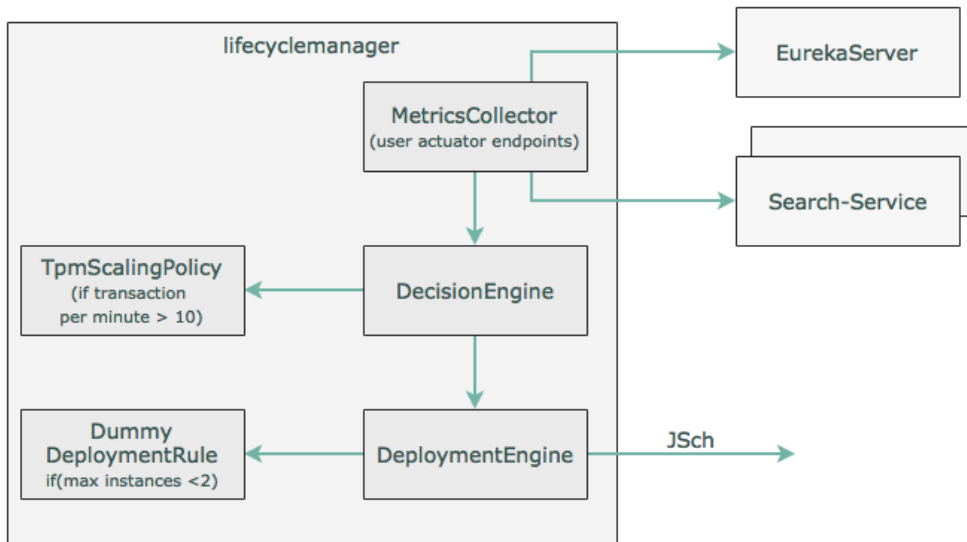
The full source code is available under the Chapter 6 project in the code files. The `chapter5.configserver`, `chapter5.eurekaclient`, `chapter5.search`, and `chapter5.search-apigateway` are copied and renamed as `chapter6.*`, respectively.

Perform the following steps to implement the custom life cycle manager:

1. Create a new Spring Boot application and name it `chapter6.lifecyclemanager`. The project structure is shown in the following diagram:



The flowchart for this example is as shown in the following diagram:



The components of this diagram are explained in details here.

2. Create a **MetricsCollector** class with the following method. At the startup of the Spring Boot application, this method will be invoked using **CommandLineRunner**, as follows:

```
public void start() {
 while(true) {
 eurekaClient.getServices().forEach(service -> { System.
out.println("discovered service " + service);
 Map metrics = restTemplate.getForObject("http://" + service + "/"
metrics", Map.class);
 decisionEngine.execute(service, metrics);
 });
 }
}
```

The preceding method looks for the services registered in the Eureka server and gets all the instances. In the real world, rather than polling, the instances should publish metrics to a common place, where metrics aggregation will happen.

3. The following `DecisionEngine` code accepts the metric and applies certain scaling policies to determine whether the service requires scaling up or not:

```
public boolean execute(String serviceId, Map metrics) {
 if (scalingPolicies.getPolicy(serviceId).
 execute(serviceId, metrics)) {
 return deploymentEngine.scaleUp(deploymentRules.
 getDeploymentRules(serviceId), serviceId);
 }
 return false;
}
```

4. Based on the service ID, the policies that are related to the services will be picked up and applied. In this case, a minimal TPM scaling policy is implemented in `TpmScalingPolicy`, as follows:

```
public class TpmScalingPolicy implements ScalingPolicy {
 public boolean execute(String serviceId, Map metrics) {
 if (metrics.containsKey("gauge.servo.tpm")) {
 Double tpm = (Double) metrics.get("gauge.servo.tpm");
 System.out.println("gauge.servo.tpm " + tpm);
 return (tpm > 10);
 }
 return false;
 }
}
```

5. If the policy returns `true`, `DecisionEngine` then invokes `DeploymentEngine` to spin up another instance. `DeploymentEngine` makes use of `DeploymentRules` to decide how to execute scaling. The rules can enforce the number of min and max instances, in which region or machine the new instance has to be started, the resources required for the new instance, and so on. `DummyDeploymentRule` simply makes sure the max instance is not more than 2.
6. `DeploymentEngine`, in this case, uses the **JSch (Java Secure Channel)** library from JCraft to SSH to the destination server and start the service. This requires the following additional Maven dependency:

```
<dependency>
 <groupId>com.jcraft</groupId>
 <artifactId>jsch</artifactId>
 <version>0.1.53</version>
</dependency>
```



7. The current SSH implementation is kept simple enough as we will change this in future chapters. In this example, `DeploymentEngine` sends the following command over the SSH library on the target machine:

```
String command ="java -jar -Dserver.port=8091 ./work/codebox/
chapter6/chapter6.search/target/search-1.0.jar";
```

Integration with Nexus happens from the target machine using Linux scripts with Nexus CLI or using `curl`. In this example, we will not explore Nexus.

8. The next step is to change the Search microservice to expose a new gauge for TPM. We have to change all the microservices developed earlier to submit this additional metric.

We will only examine Search in this chapter, but in order to complete it, all the services have to be updated. In order to get the `gauge.servo.tpm` metrics, we have to add `TPMCounter` to all the microservices.

The following code counts the transactions over a sliding window of 1 minute:

```
class TPMCounter {
 LongAdder count;
 Calendar expiry = null;
 TPMCounter(){
 reset();
 }
 void reset () {
 count = new LongAdder();
 expiry = Calendar.getInstance();
 expiry.add(Calendar.MINUTE, 1);
 }
 boolean isExpired(){
 return Calendar.getInstance().after(expiry);
 }
 void increment(){
 if(isExpired()){
 reset();
 }
 count.increment();
 }
}
```

9. The following code needs to be added to `SearchController` to set the `tpm` value:

```
class SearchRestController {
 TPMCounter tpm = new TPMCounter();
 @Autowired
 GaugeService gaugeService;
 //other code
```

10. The following code is from the get REST endpoint (the search method) of `SearchRestController`, which submits the `tpm` value as a gauge to the actuator endpoint:

```
tpm.increment();
gaugeService.submit("tpm", tpm.count.intValue());
```

## Running the life cycle manager

Perform the following steps to run the life cycle manager developed in the previous section:

1. Edit `DeploymentEngine.java` and update the password to reflect the machine's password, as follows. This is required for the SSH connection:

```
session.setPassword("rajeshrv");
```

2. Build all the projects by running Maven from the root folder (Chapter 6) via the following command:

```
mvn -Dmaven.test.skip=true clean install
```

3. Then, run RabbitMQ, as follows:

```
./rabbitmq-server
```

4. Ensure that the Config server is pointing to the right configuration repository. We need to add a property file for the life cycle manager
5. Run the following commands from the respective project folders:

```
java -jar target/config-server-0.0.1-SNAPSHOT.jar
```

```
java -jar target/eureka-server-0.0.1-SNAPSHOT.jar
```

```
java -jar target/lifecycle-manager-0.0.1-SNAPSHOT.jar
```

```
java -jar target/search-1.0.jar
```

```
java -jar target/search-apigateway-1.0.jar
```

```
java -jar target/website-1.0.jar
```

6. Once all the services are started, open a browser window and load `http://localhost:8001`.
7. Execute the flight search 11 times, one after the other, within a minute. This will trigger the decision engine to instantiate another instance of the Search microservice.
8. Open the Eureka console (`http://localhost:8761`) and watch for a second **SEARCH-SERVICE**. Once the server is started, the instances will appear as shown here:

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones	Status
LIFECYCLE-MANAGER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.106:lifecycle-manager-service:9090
SEARCH-APIGATEWAY	n/a (1)	(1)	UP (1) - 192.168.0.106:search-apigateway:8095
SEARCH-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.106:search-service:8091 , 192.168.0.106:search-service:8090
TEST-CLIENT	n/a (1)	(1)	UP (1) - 192.168.0.106:test-client:8001

## Summary

In this chapter, you learned the importance of autoscaling when deploying large-scale microservices.

We also explored the concept of autoscaling and the different models of and approaches to autoscaling, such as the time-based, resource-based, queue length-based, and predictive ones. We then reviewed the role of a life cycle manager in the context of microservices and reviewed its capabilities. Finally, we ended this chapter by reviewing a sample implementation of a simple custom life cycle manager in the context of BrownField PSS microservices.

Autoscaling is an important supporting capability required when dealing with large-scale microservices. We will discuss a more mature implementation of the life cycle manager in *Chapter 9, Managing Dockerized Microservices with Mesos and Marathon*.

The next chapter will explore the logging and monitoring capabilities that are indispensable for successful microservice deployments.

# 7

## Logging and Monitoring Microservices

One of the biggest challenges due to the very distributed nature of Internet-scale microservices deployment is the logging and monitoring of individual microservices. It is difficult to trace end-to-end transactions by correlating logs emitted by different microservices. As with monolithic applications, there is no single pane of glass to monitor microservices.

This chapter will cover the necessity and importance of logging and monitoring in microservice deployments. This chapter will further examine the challenges and solutions to address logging and monitoring with a number of potential architectures and technologies.

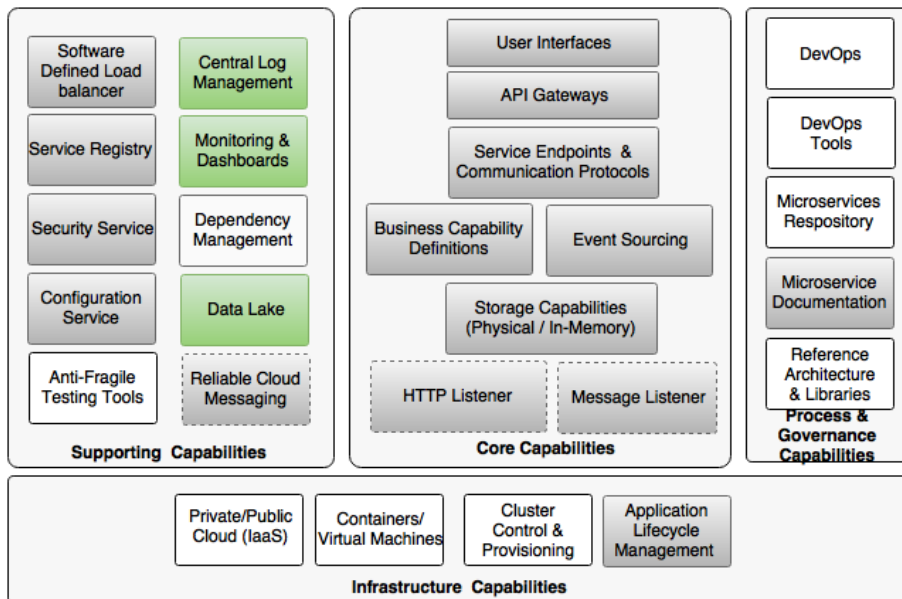
By the end of this chapter, you will learn about:

- The different options, tools, and technologies for log management
- The use of Spring Cloud Sleuth in tracing microservices
- The different tools for end-to-end monitoring of microservices
- The use of Spring Cloud Hystrix and Turbine for circuit monitoring
- The use of data lakes in enabling business data analysis

# Reviewing the microservice capability model

In this chapter, we will explore the following microservice capabilities from the microservices capability model discussed in *Chapter 3, Applying Microservices Concepts*:

- **Central Log Management**
- **Monitoring and Dashboards**
- **Dependency Management** (part of Monitoring and Dashboards)
- **Data Lake**



## Understanding log management challenges

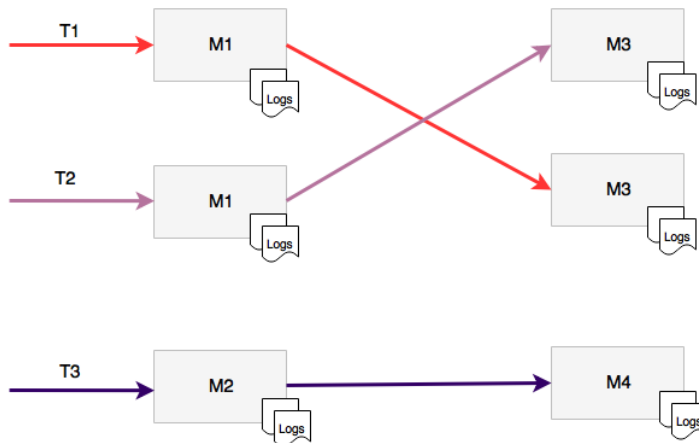
Logs are nothing but streams of events coming from a running process. For traditional JEE applications, a number of frameworks and libraries are available to log. Java Logging (JUL) is an option off the shelf from Java itself. Log4j, Logback, and SLF4J are some of the other popular logging frameworks available. These frameworks support both UDP as well as TCP protocols for logging. Applications send log entries to the console or to the filesystem. File recycling techniques are generally employed to avoid logs filling up all the disk space

One of the best practices of log handling is to switch off most of the log entries in production due to the high cost of disk IOs. Not only do disk IOs slow down the application, but they can also severely impact scalability. Writing logs into the disk also requires high disk capacity. An out-of-disk-space scenario can bring down the application. Logging frameworks provide options to control logging at runtime to restrict what is to be printed and what not. Most of these frameworks provide fine grained control over the logging controls. They also provide options to change these configurations at runtime

On the other hand, logs may contain important information and have high value if properly analyzed. Therefore, restricting log entries essentially limits our ability to understand the application's behavior.

When moved from traditional to cloud deployment, applications are no longer locked to a particular, predefined machine. Virtual machines and containers are not hardwired with an application. The machines used for deployment can change from time to time. Moreover, containers such as Docker are ephemeral. This essentially means that one cannot rely on the persistent state of the disk. Logs written to the disk are lost once the container is stopped and restarted. Therefore, we cannot rely on the local machine's disk to write log files

As we discussed in *Chapter 1, Demystifying Microservices*, one of the principles of the Twelve-Factor app is to avoid routing or storing log files by the application itself. In the context of microservices, they will run on isolated physical or virtual machines, resulting in fragmented log files. In this case, it is almost impossible to trace end-to-end transactions that span multiple microservices:



As shown in the diagram, each microservice emits logs to a local filesystem. In this case, microservice M1 calls M3. These services write their logs to their own local filesystems. This makes it harder to correlate and understand the end-to-end transaction flow. Also, as shown in the diagram, there are two instances of M1 and two instances of M2 running on two different machines. In this case, log aggregation at the service level is hard to achieve.

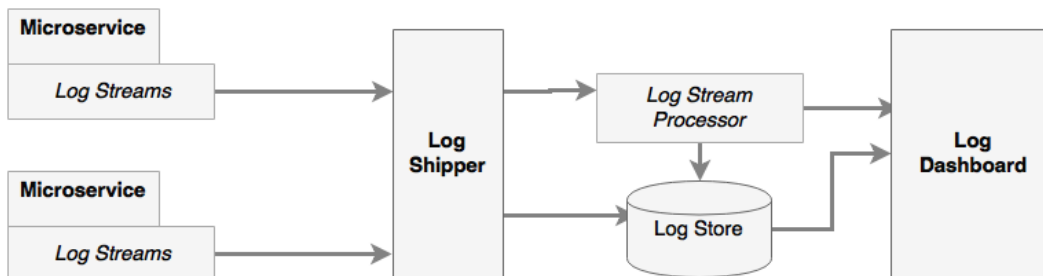
## A centralized logging solution

In order to address the challenges stated earlier, traditional logging solutions require serious rethinking. The new logging solution, in addition to addressing the preceding challenges, is also expected to support the capabilities summarized here:

- The ability to collect all log messages and run analytics on top of the log messages
- The ability to correlate and track transactions end to end
- The ability to keep log information for longer time periods for trending and forecasting
- The ability to eliminate dependency on the local disk system
- The ability to aggregate log information coming from multiple sources such as network devices, operating system, microservices, and so on

The solution to these problems is to centrally store and analyze all log messages, irrespective of the source of log. The fundamental principle employed in the new logging solution is to detach log storage and processing from service execution environments. Big data solutions are better suited to storing and processing large numbers of log messages more effectively than storing and processing them in microservice execution environments.

In the centralized logging solution, log messages will be shipped from the execution environment to a central big data store. Log analysis and processing will be handled using big data solutions:



As shown in the preceding logical diagram, there are a number of components in the centralized logging solution, as follows:

- **Log streams:** These are streams of log messages coming out of source systems. The source system can be microservices, other applications, or even network devices. In typical Java-based systems, these are equivalent to streaming Log4j log messages.
- **Log shippers:** Log shippers are responsible for collecting the log messages coming from different sources or endpoints. The log shippers then send these messages to another set of endpoints, such as writing to a database, pushing to a dashboard, or sending it to stream-processing endpoint for further real-time processing.
- **Log store:** A log store is the place where all log messages are stored for real-time analysis, trending, and so on. Typically, a log store is a NoSQL database, such as HDFS, capable of handling large data volumes.
- **Log stream processor:** The log stream processor is capable of analyzing real-time log events for quick decision making. A stream processor takes actions such as sending information to a dashboard, sending alerts, and so on. In the case of self-healing systems, stream processors can even take actions to correct the problems.
- **Log dashboard:** A dashboard is a single pane of glass used to display log analysis results such as graphs and charts. These dashboards are meant for the operational and management staff.

The benefit of this centralized approach is that there is no local I/O or blocking disk writes. It also does not use the local machine's disk space. This architecture is fundamentally similar to the lambda architecture for big data processing.



To read more on the Lambda architecture, go to <http://lambda-architecture.net>.



It is important to have in each log message a context, message, and correlation ID. The context typically has the timestamp, IP address, user information, process details (such as service, class, and functions), log type, classification, and so on. The message will be plain and simple free text information. The correlation ID is used to establish the link between service calls so that calls spanning microservices can be traced.



# The selection of logging solutions

There are a number of options available to implement a centralized logging solution. These solutions use different approaches, architectures, and technologies. It is important to understand the capabilities required and select the right solution that meets the needs.

## Cloud services

There are a number of cloud logging services available, such as the SaaS solution.

Loggly is one of the most popular cloud-based logging services. Spring Boot microservices can use Loggly's Log4j and Logback appenders to directly stream log messages into the Loggly service.

If the application or service is deployed in AWS, AWS CloudTrail can be integrated with Loggly for log analysis.

Papertrail, Logsene, Sumo Logic, Google Cloud Logging, and Logentries are examples of other cloud-based logging solutions.

The cloud logging services take away the overhead of managing complex infrastructures and large storage solutions by providing them as simple-to-integrate services. However, latency is one of the key factors to be considered when selecting cloud logging as a service.

## Off-the-shelf solutions

There are many purpose-built tools to provide end-to-end log management capabilities that are installable locally in an on-premises data center or in the cloud.

Graylog is one of the popular open source log management solutions. Graylog uses Elasticsearch for log storage and MongoDB as a metadata store. Graylog also uses GELF libraries for Log4j log streaming.

Splunk is one of the popular commercial tools available for log management and analysis. Splunk uses the log file shipping approach, compared to log streaming used by other solutions to collect logs.

## Best-of-breed integration

The last approach is to pick and choose best-of-breed components and build a custom logging solution.

## Log shippers

There are log shippers that can be combined with other tools to build an end-to-end log management solution. The capabilities differ between different log shipping tools.

Logstash is a powerful data pipeline tool that can be used to collect and ship log files. Logstash acts as a broker that provides a mechanism to accept streaming data from different sources and sync them to different destinations. Log4j and Logback appenders can also be used to send log messages directly from Spring Boot microservices to Logstash. The other end of Logstash is connected to Elasticsearch, HDFS, or any other database.

Fluentd is another tool that is very similar to Logstash, as is Logspout, but the latter is more appropriate in a Docker container-based environment.

## Log stream processors

Stream-processing technologies are optionally used to process log streams on the fly. For example, if a 404 error is continuously occurring as a response to a particular service call, it means there is something wrong with the service. Such situations have to be handled as soon as possible. Stream processors are pretty handy in such cases as they are capable of reacting to certain streams of events that a traditional reactive analysis can't.

A typical architecture used for stream processing is a combination of Flume and Kafka together with either Storm or Spark Streaming. Log4j has Flume appenders, which are useful to collect log messages. These messages are pushed into distributed Kafka message queues. The stream processors collect data from Kafka and process them on the fly before sending it to Elasticsearch and other log stores

Spring Cloud Stream, Spring Cloud Stream Modules, and Spring Cloud Data Flow can also be used to build the log stream processing.

## Log storage

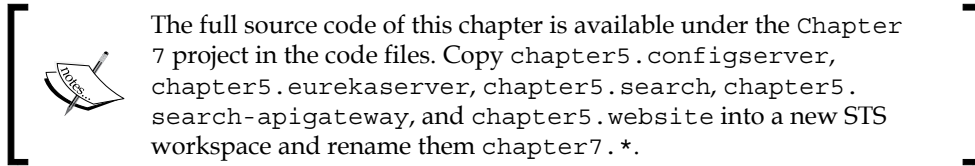
Real-time log messages are typically stored in Elasticsearch. Elasticsearch allows clients to query based on text-based indexes. Apart from Elasticsearch, HDFS is also commonly used to store archived log messages. MongoDB or Cassandra is used to store summary data, such as monthly aggregated transaction counts. Offline log processing can be done using Hadoop's MapReduce programs.

## Dashboards

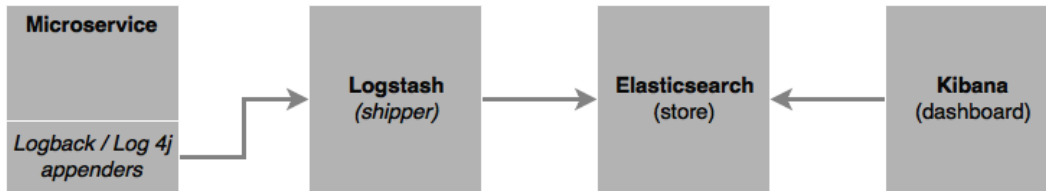
The last piece required in the central logging solution is a dashboard. The most commonly used dashboard for log analysis is Kibana on top of an Elasticsearch data store. Graphite and Grafana are also used to display log analysis reports.

## A custom logging implementation

The tools mentioned before can be leveraged to build a custom end-to-end logging solution. The most commonly used architecture for custom log management is a combination of Logstash, Elasticsearch, and Kibana, also known as the ELK stack.



The following diagram shows the log monitoring flow



In this section, a simple implementation of a custom logging solution using the ELK stack will be examined.

Follow these steps to implement the ELK stack for logging:

1. Download and install Elasticsearch, Kibana, and Logstash from <https://www.elastic.co>.
2. Update the Search microservice (`chapter7.search`). Review and ensure that there are some log statements in the Search microservice. The log statements are nothing special but simple log statements using `slf4j`, as follows:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
//other code goes here
private static final Logger logger = LoggerFactory.
 getLogger(SearchRestController.class);
//other code goes here
```

```

logger.info("Looking to load flights...");
for (Flight flight : flightRepository.
 findByOriginAndDestinationAndFlightDate
 ("NYC", "SFO", "22-JAN-16")) {
 logger.info(flight.toString());
}

```

3. Add the logstash dependency to integrate logback to Logstash in the Search service's pom.xml file, as follows

```

<dependency>
 <groupId>net.logstash.logback</groupId>
 <artifactId>logstash-logback-encoder</artifactId>
 <version>4.6</version>
</dependency>

```

4. Also, downgrade the logback version to be compatible with Spring 1.3.5.RELEASE via the following line:

```

<logback.version>1.1.6</logback.version>

```

5. Override the default Logback configuration. This can be done by adding a new logback.xml file under src/main/resources, as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <include resource="org/springframework/boot/logging/logback/
defaults.xml"/>
 <include resource="org/springframework/boot/logging/logback/
console-appender.xml" />
 <appender name="stash" class="net.logstash.logback.
appender.LogstashTcpSocketAppender">
 <destination>localhost:4560</destination>
 <!-- encoder is required -->
 <encoder class="net.logstash.logback.encoder.
LogstashEncoder" />
 </appender>
 <root level="INFO">
 <appender-ref ref="CONSOLE" />
 <appender-ref ref="stash" />
 </root>
</configuration>

```

The preceding configuration overrides the default Logback configuration by adding a new TCP socket appender, which streams all the log messages to a Logstash service, which is listening on port 4560. It is important to add an encoder, as mentioned in the previous configuration.

6. Create a configuration as shown in the following code and store it in a `logstash.conf` file. The location of this file is irrelevant as it will be passed as an argument when starting Logstash. This configuration will take input from the socket listening on 4560 and send the output to Elasticsearch running on 9200. The `stdout` is optional and is set to debug:

```
input {
 tcp {
 port => 4560
 host => localhost
 }
}
output {
 elasticsearch { hosts => ["localhost:9200"] }
 stdout { codec => rubydebug }
}
```

7. Run Logstash, Elasticsearch, and Kibana from their respective installation folders, as follows:  

```
./bin/logstash -f logstash.conf
```

```
./bin/elasticsearch
```

```
./bin/kibana
```
8. Run the Search microservice. This will invoke the unit test cases and result in printing the log statements mentioned before.
9. Go to a browser and access Kibana, at `http://localhost:5601`.
10. Go to **Settings | Configure an index pattern**, as shown here:

## Configure an index pattern

In order to use Kibana you must configure at least one index pattern. Index patterns are used to identify the Elasticsearch index to run search and analytics against. They are also used to configure fields.

☒ Index contains time-based events  
☐ Use event times to create index names

**Index name or pattern**  
Patterns allow you to define dynamic index names using \* as a wildcard. Example: logstash-\*

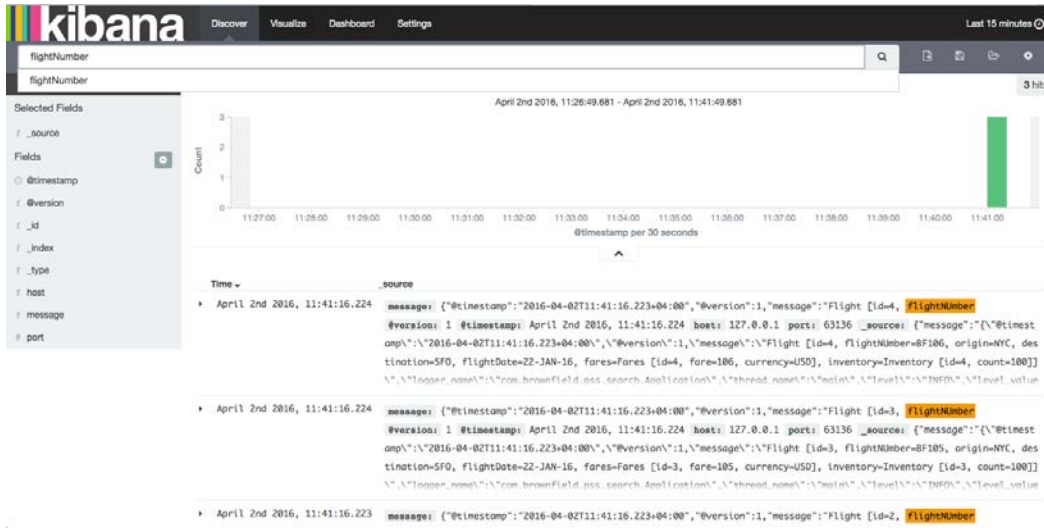
logstash-\*

**Time-field name** ⓘ refresh fields  
@timestamp

Create

- Go to the **Discover** menu to see the logs. If everything is successful, we will see the Kibana screenshot as follows. Note that the log messages are displayed in the Kibana screen.

Kibana provides out-of-the-box features to build summary charts and graphs using log messages:



## Distributed tracing with Spring Cloud Sleuth

The previous section addressed microservices' distributed and fragmented logging issue by centralizing the log data. With the central logging solution, we can have all the logs in a central storage. However, it is still almost impossible to trace end-to-end transactions. In order to do end-to-end tracking, transactions spanning microservices need to have a correlation ID.

Twitter's Zipkin, Cloudera's HTrace, and Google's Dapper systems are examples of distributed tracing systems. Spring Cloud provides a wrapper component on top of these using the Spring Cloud Sleuth library.

Distributed tracing works with the concepts of **span** and **trace**. The span is a unit of work; for example, calling a service is identified by a 64-bit span ID. A set of spans form a tree-like structure is called a trace. Using the trace ID, the call can be tracked end to end:



As shown in the diagram, **Microservice 1** calls **Microservice 2**, and **Microservice 2** calls **Microservice 3**. In this case, as shown in the diagram, the same trace ID is passed across all microservices, which can be used to track transactions end to end.

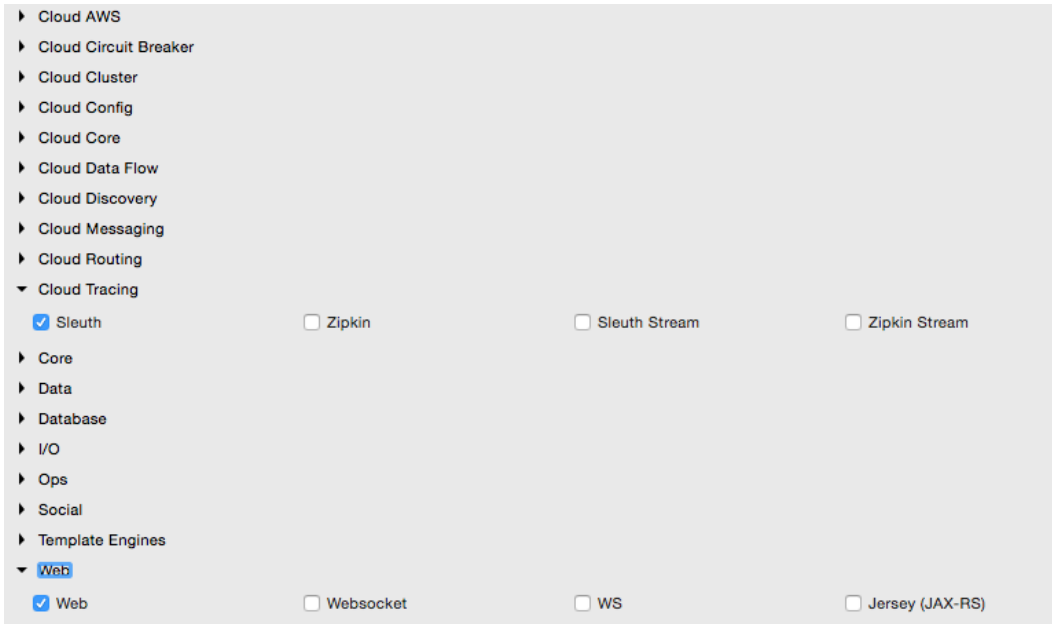
In order to demonstrate this, we will use the Search API Gateway and Search microservices. A new endpoint has to be added in Search API Gateway (chapter7. search-apigateway) that internally calls the Search service to return data. Without the trace ID, it is almost impossible to trace or link calls coming from the Website to Search API Gateway to Search microservice. In this case, it only involves two to three services, whereas in a complex environment, there could be many interdependent services.

Follow these steps to create the example using Sleuth:

1. Update Search and Search API Gateway. Before this, the Sleuth dependency needs to be added to the respective POM files, which can be done via the following code:

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

2. In the case of building a new service, select **Sleuth** and **Web**, as shown here:



3. Add the Logstash dependency to the Search service as well as the Logback configuration, as in the previous example
4. The next step is to add two more properties in the Logback configuration, as follows:

```
<property name="spring.application.name" value="search-service"/>
<property name="CONSOLE_LOG_PATTERN" value="%d{yyyy-MM-dd
HH:mm:ss.SSS} [{spring.application.name}] [trace=%X{X-Trace-Id:-
},span=%X{X-Span-Id:-}] [%15.15t] %-40.40logger{39}: %m%n"/>
```

The first property is the name of the application. The names given in this are the service IDs: `search-service` and `search-apigateway` in Search and Search API Gateway, respectively. The second property is an optional pattern used to print the console log messages with a trace ID and span ID. The preceding change needs to be applied to both the services.

5. Add the following piece of code to advise Sleuth when to start a new span ID in the Spring Boot Application class. In this case, `AlwaysSampler` is used to indicate that the span ID has to be created every time a call hits the service. This change needs to be applied in both the services:

```
@Bean
public AlwaysSampler defaultSampler() {
 return new AlwaysSampler();
}
```



6. Add a new endpoint to Search API Gateway, which will call the Search service as follows. This is to demonstrate the propagation of the trace ID across multiple microservices. This new method in the gateway returns the operating hub of the airport by calling the Search service, as follows:

```
@RequestMapping("/hubongw")
String getHub(HttpServletRequest req){
 logger.info("Search Request in API gateway for getting Hub,
forwarding to search-service ");
 String hub = restTemplate.getForObject("http://search-service/
search/hub", String.class);
 logger.info("Response for hub received, Hub "+ hub);
 return hub;
}
```

7. Add another endpoint in the Search service, as follows:

```
@RequestMapping("/hub")
String getHub(){
 logger.info("Searching for Hub, received from search-
apigateway ");
 return "SFO";
}
```

8. Once added, run both the services. Hit the gateway's new hub on the gateway (/hubongw) endpoint using a browser (http://localhost:8095/hubongw).

As mentioned earlier, the Search API Gateway service is running on 8095 and the Search service is running on 8090.

9. Look at the console logs to see the trace ID and span IDs printed. The first print is from Search API Gateway, and the second one came from the Search service. Note that the trace IDs are the same in both the cases, as follows:

```
2016-04-02 17:24:37.624 [search-apigateway] [trace=8a7e278f-7b2b-
43e3-a45c-69d3ca66d663, span=8a7e278f-7b2b-43e3-a45c-69d3ca66d663]
[io-8095-exec-10] c.b.p.s.a.SearchAPIGatewayController :
Response for hub received, Hub DXB
```

```
2016-04-02 17:24:37.612 [search-service] [trace=8a7e278f-7b2b-
43e3-a45c-69d3ca66d663, span=fd309bba-5b4d-447f-a5e1-7faaab90cfb1]
[nio-8090-exec-1] c.b.p.search.component.SearchComponent :
Searching for Hub, received from search-apigateway
```



One of the main objectives of microservice monitoring is to understand the behavior of the system from a user experience point of view. This will ensure that the end-to-end behavior is consistent and is in line with what is expected by the users.

## Monitoring challenges

Similar to the fragmented logging issue, the key challenge in monitoring microservices is that there are many moving parts in a microservice ecosystem.

The typical issues are summarized here:

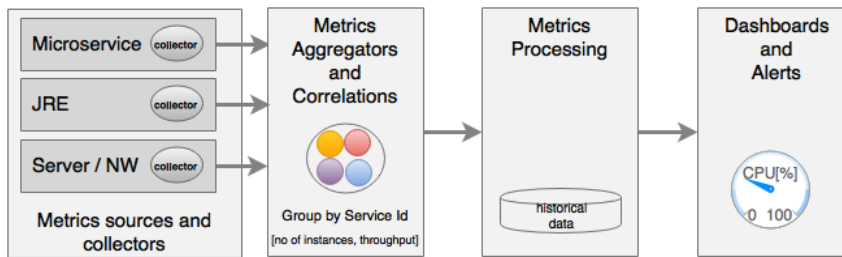
- The statistics and metrics are fragmented across many services, instances, and machines.
- Heterogeneous technologies may be used to implement microservices, which makes things even more complex. A single monitoring tool may not give all the required monitoring options.
- Microservices deployment topologies are dynamic, making it impossible to preconfigure servers, instances, and monitoring parameters.

Many of the traditional monitoring tools are good to monitor monolithic applications but fall short in monitoring large-scale, distributed, interlinked microservice systems. Many of the traditional monitoring systems are agent-based preinstall agents on the target machines or application instances. This poses two challenges:

- If the agents require deep integration with the services or operating systems, then this will be hard to manage in a dynamic environment
- If these tools impose overheads when monitoring or instrumenting the application, it may lead to performance issues

Many traditional tools need baseline metrics. Such systems work with preset rules, such as if the CPU utilization goes above 60% and remains at this level for 2 minutes, then an alert should be sent to the administrator. It is extremely hard to preconfigure these values in large, Internet-scale deployments.

New-generation monitoring applications learn the application's behavior by themselves and set automatic threshold values. This frees up administrators from doing this mundane task. Automated baselines are sometimes more accurate than human forecasts:



As shown in the diagram, the key areas of microservices monitoring are:

- **Metrics sources and data collectors:** Metrics collection at the source is done either by the server pushing metrics information to a central collector or by embedding lightweight agents to collect information. Data collectors collect monitoring metrics from different sources, such as network, physical machines, containers, software components, applications, and so on. The challenge is to collect this data using autodiscovery mechanisms instead of static configurations.

This is done by either running agents on the source machines, streaming data from the sources, or polling at regular intervals.

- **Aggregation and correlation of metrics:** Aggregation capability is required for aggregating metrics collected from different sources, such as user transaction, service, infrastructure, network, and so on. Aggregation can be challenging as it requires some level of understanding of the application's behavior, such as service dependencies, service grouping, and so on. In many cases, these are automatically formulated based on the metadata provided by the sources.

Generally, this is done by an intermediary that accept the metrics.

- **Processing metrics and actionable insights:** Once data is aggregated, the next step is to do the measurement. Measurements are typically done using set thresholds. In the new-generation monitoring systems, these thresholds are automatically discovered. Monitoring tools then analyze the data and provide actionable insights.

These tools may use big data and stream analytics solutions.

- **Alerting, actions, and dashboards:** As soon as issues are detected, they have to be notified to the relevant people or systems. Unlike traditional systems, the microservices monitoring systems should be capable of taking actions on a real-time basis. Proactive monitoring is essential to achieving self-healing. Dashboards are used to display SLAs, KPIs, and so on.

Dashboards and alerting tools are capable of handling these requirements.

Microservice monitoring is typically done with three approaches. A combination of these is really required for effective monitoring:

- **Application performance monitoring (APM):** This is more of a traditional approach to system metrics collection, processing, alerting, and dashboard rendering. These are more from the system's point of view. Application topology discovery and visualization are new capabilities implemented by many of the APM tools. The capabilities vary between different APM providers.
- **Synthetic monitoring:** This is a technique that is used to monitor the system's behavior using end-to-end transactions with a number of test scenarios in a production or production-like environment. Data is collected to validate the system's behavior and potential hotspots. Synthetic monitoring helps understand the system dependencies as well.
- **Real user monitoring (RUM) or user experience monitoring:** This is typically a browser-based software that records real user statistics, such as response time, availability, and service levels. With microservices, with more frequent release cycle and dynamic topology, user experience monitoring is more important.

## Monitoring tools

There are many tools available to monitor microservices. There are also overlaps between many of these tools. The selection of monitoring tools really depends upon the ecosystem that needs to be monitored. In most cases, more than one tool is required to monitor the overall microservice ecosystem.

The objective of this section is to familiarize ourselves with a number of common microservices-friendly monitoring tools:

- AppDynamics, Dynatrace, and New Relic are top commercial vendors in the APM space, as per Gartner Magic Quadrant 2015. These tools are microservice friendly and support microservice monitoring effectively in a single console. Ruxit, Datadog, and Dataloop are other commercial offerings that are purpose-built for distributed systems that are essentially microservices friendly. Multiple monitoring tools can feed data to Datadog using plugins.
- Cloud vendors come with their own monitoring tools, but in many cases, these monitoring tools alone may not be sufficient for large-scale microservice monitoring. For instance, AWS uses CloudWatch and Google Cloud Platform uses Cloud Monitoring to collect information from various sources.

- Some of the data collecting libraries, such as Zabbix, statd, collectd, jmxtrans, and so on operate at a lower level in collecting runtime statistics, metrics, gauges, and counters. Typically, this information is fed into data collectors and processors such as Riemann, Datadog, and Librato, or dashboards such as Graphite.
- Spring Boot Actuator is one of the good vehicles to collect microservices metrics, gauges, and counters, as we discussed in *Chapter 2, Building Microservices with Spring Boot*. Netflix Servo, a metric collector similar to Actuator, and the QBit and Dropwizard metrics also fall in the same category of metric collectors. All these metrics collectors need an aggregator and dashboard to facilitate full-sized monitoring.
- Monitoring through logging is popular but a less effective approach in microservices monitoring. In this approach, as discussed in the previous section, log messages are shipped from various sources, such as microservices, containers, networks, and so on to a central location. Then, we can use the logs files to trace transactions, identify hotspots, and so on. Loggly, ELK, Splunk, and Trace are candidates in this space.
- Sensu is a popular choice for microservice monitoring from the open source community. Weave Scope is another tool, primarily targeting containerized deployments. Spigo is one of the purpose-built microservices monitoring systems closely aligned with the Netflix stack.
- Pingdom, New Relic Synthetics, Runscope, Catchpoint, and so on provide options for synthetic transaction monitoring and user experience monitoring on live systems.
- Circonus is classified more as a DevOps monitoring tool but can also do microservices monitoring. Nagios is a popular open source monitoring tool but falls more into the traditional monitoring system.
- Prometheus provides a time series database and visualization GUI useful in building custom monitoring tools.

## Monitoring microservice dependencies

When there are a large number of microservices with dependencies, it is important to have a monitoring tool that can show the dependencies among microservices. It is not a scalable approach to statically configure and manage these dependencies. There are many tools that are useful in monitoring microservice dependencies, as follows:

- Mentoring tools such as AppDynamics, Dynatrace, and New Relic can draw dependencies among microservices. End-to-end transaction monitoring can also trace transaction dependencies. Other monitoring tools, such as Spigo, are also useful for microservices dependency management.

- CMDB tools such as Device42 or purpose-built tools such as Accordance are useful in managing the dependency of microservices. **Veritas Risk Advisor (VRA)** is also useful for infrastructure discovery.
- A custom implementation with a Graph database, such as Neo4j, is also useful. In this case, a microservice has to preconfigure its direct and indirect dependencies. At the startup of the service, it publishes and cross-checks its dependencies with a Neo4j database.

## Spring Cloud Hystrix for fault-tolerant microservices

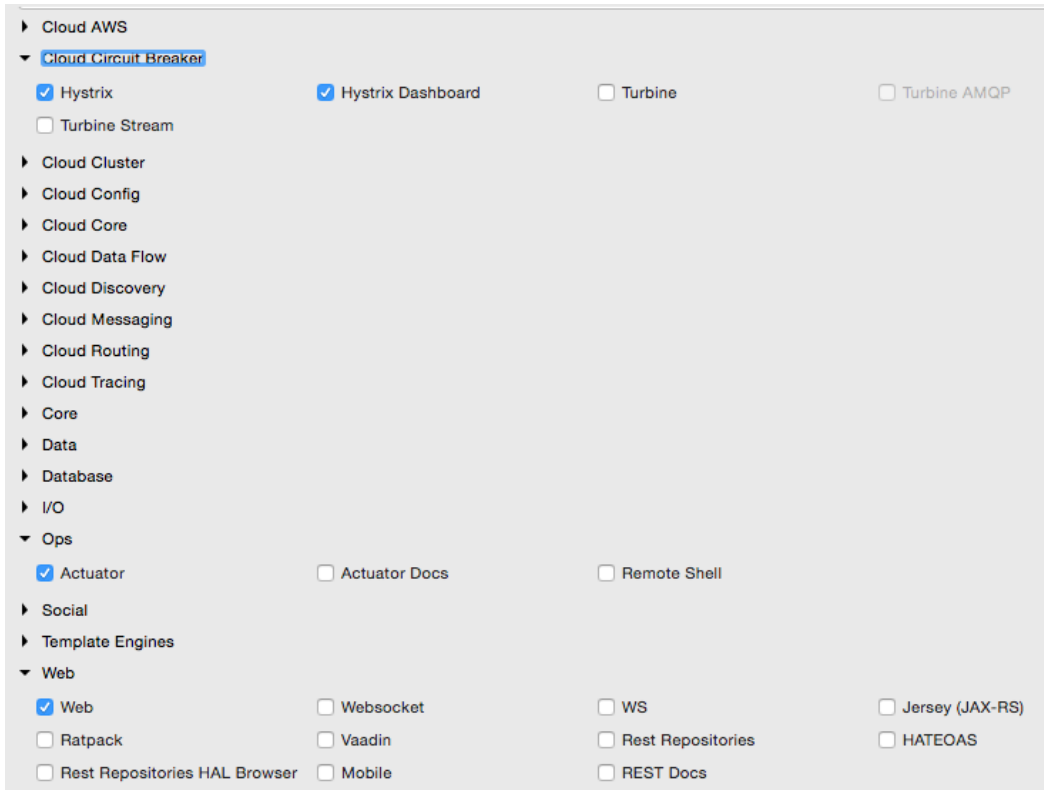
This section will explore Spring Cloud Hystrix as a library for a fault-tolerant and latency-tolerant microservice implementation. Hystrix is based on the *fail fast* and *rapid recovery* principles. If there is an issue with a service, Hystrix helps isolate it. It helps to recover quickly by falling back to another preconfigured fallback service. Hystrix is another battle-tested library from Netflix. Hystrix is based on the circuit breaker pattern.



Read more about the circuit breaker pattern at <https://msdn.microsoft.com/en-us/library/dn589784.aspx>.

In this section, we will build a circuit breaker with Spring Cloud Hystrix. Perform the following steps to change the Search API Gateway service to integrate it with Hystrix:

1. Update the Search API Gateway service. Add the Hystrix dependency to the service. If developing from scratch, select the following libraries:



2. In the Spring Boot Application class, add `@EnableCircuitBreaker`. This command will tell Spring Cloud Hystrix to enable a circuit breaker for this application. It also exposes the `/hystrix.stream` endpoint for metrics collection.
3. Add a component class to the Search API Gateway service with a method; in this case, this is `getHub` annotated with `@HystrixCommand`. This tells Spring that this method is prone to failure. Spring Cloud libraries wrap these methods to handle fault tolerance and latency tolerance by enabling circuit breaker. The Hystrix command typically follows with a fallback method. In case of failure, Hystrix automatically enables the fallback method mentioned and diverts traffic to the fallback method. As shown in the following code, in this case, `getHub` will fall back to `getDefaultHub`:

```
@Component
class SearchAPIGatewayComponent {
 @LoadBalanced
 @Autowired
 RestTemplate restTemplate;
```



```
@HystrixCommand(fallbackMethod = "getDefaultHub")
public String getHub() {
 String hub = restTemplate.getForObject("http://search-service/
search/hub", String.class);
 return hub;
}
public String getDefaultHub() {
 return "Possibly SFO";
}
}
```

4. The `getHub` method of `SearchAPIGatewayController` calls the `getHub` method of `SearchAPIGatewayComponent`, as follows:

```
@RequestMapping("/hubongw")
String getHub() {
 logger.info("Search Request in API gateway for getting Hub,
forwarding to search-service ");
 return component.getHub();
}
```

5. The last part of this exercise is to build a Hystrix Dashboard. For this, build another Spring Boot application. Include Hystrix, Hystrix Dashboard, and Actuator when building this application.
6. In the Spring Boot Application class, add the `@EnableHystrixDashboard` annotation.
7. Start the Search service, Search API Gateway, and Hystrix Dashboard applications. Point the browser to the Hystrix Dashboard application's URL. In this example, the Hystrix Dashboard is started on port 9999. So, open the URL `http://localhost:9999/hystrix`.
8. A screen similar to the following screenshot will be displayed. In the Hystrix Dashboard, enter the URL of the service to be monitored.

In this case, Search API Gateway is running on port 8095. Hence, the `hystrix.stream` URL will be `http://localhost:8095/hytrix.stream`, as shown:



## Hystrix Dashboard

<http://localhost:8095/hystrix.stream>

Cluster via Turbine (default cluster): <http://turbine-hostname:port/turbine.stream>

Cluster via Turbine (custom cluster): [http://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](http://turbine-hostname:port/turbine.stream?cluster=[clusterName])

Single Hystrix App: <http://hystrix-app:port/hystrix.stream>

Delay:  ms Title:

[Monitor Stream](#)

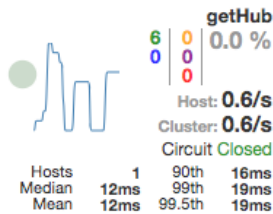
9. The Hystrix Dashboard will be displayed as follows:

### Hystrix Stream: SearchAPIGateway



**HYSTRIX**  
DEFEND YOUR APP

**Circuit** Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)  
[Success](#) | [Short-Circuited](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | [Error %](#)




**Thread Pools** Sort: [Alphabetical](#) | [Volume](#) |

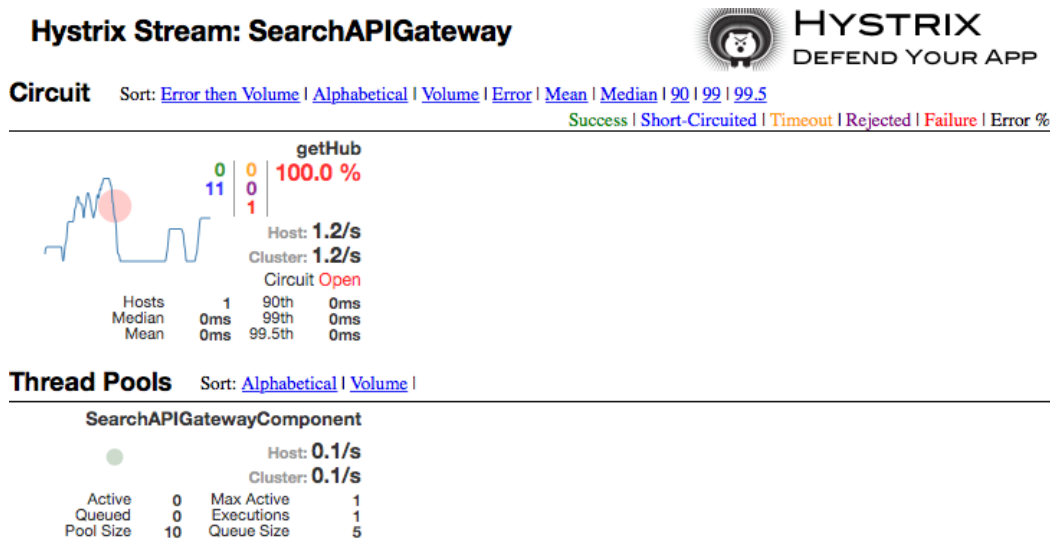
#### SearchAPIGatewayComponent


Host: 0.6/s  
Cluster: 0.6/s

Active	0	Max Active	1
Queued	0	Executions	6
Pool Size	10	Queue Size	5

 Note that at least one transaction has to be executed to see the display. This can be done by hitting `http://localhost:8095/hubongw`.

- 10. Create a failure scenario by shutting down the Search service. Note that the fallback method will be called when hitting the URL `http://localhost:8095/hubongw`.
- 11. If there are continuous failures, then the circuit status will be changed to open. This can be done by hitting the preceding URL a number of times. In the open state, the original service will no longer be checked. The Hystrix Dashboard will show the status of the circuit as **Open**, as shown in the following screenshot. Once a circuit is opened, periodically, the system will check for the original service status for recovery. When the original service is back, the circuit breaker will fall back to the original service and the status will be set to **Closed**:

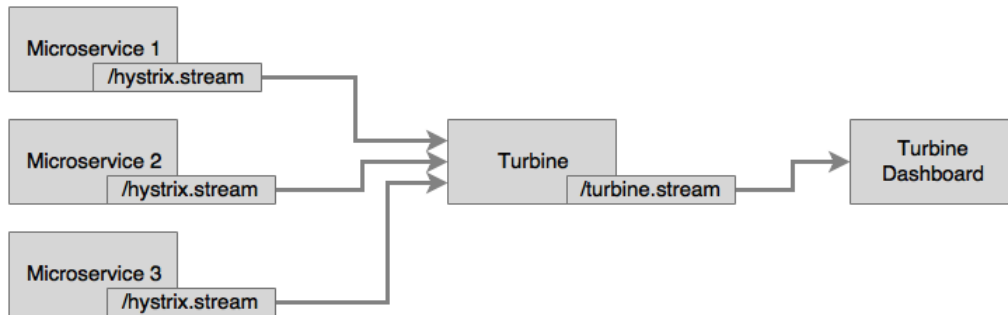


 To know the meaning of each of these parameters, visit the Hystrix wiki at <https://github.com/Netflix/Hystrix/wiki/Dashboard>.

## Aggregating Hystrix streams with Turbine

In the previous example, the `/hystrix.stream` endpoint of our microservice was given in the Hystrix Dashboard. The Hystrix Dashboard can only monitor one microservice at a time. If there are many microservices, then the Hystrix Dashboard pointing to the service has to be changed every time we switch the microservices to monitor. Looking into one instance at a time is tedious, especially when there are many instances of a microservice or multiple microservices.

We have to have a mechanism to aggregate data coming from multiple `/hystrix.stream` instances and consolidate it into a single dashboard view. Turbine does exactly the same thing. Turbine is another server that collects Hystrix streams from multiple instances and consolidates them into one `/turbine.stream` instance. Now, the Hystrix Dashboard can point to `/turbine.stream` to get the consolidated information:



Turbine currently works only with different hostnames. Each instance has to be run on separate hosts. If you are testing multiple services locally on the same host, then update the host file `/etc/hosts` to simulate multiple hosts. Once done, `bootstrap.properties` has to be configured as follows

```
eureka.instance.hostname: localdomain2
```

This example showcases how to use Turbine to monitor circuit breakers across multiple instances and services. We will use the Search service and Search API Gateway in this example. Turbine internally uses Eureka to resolve service IDs that are configured for monitoring

Perform the following steps to build and execute this example:

1. The Turbine server can be created as just another Spring Boot application using Spring Boot Starter. Select Turbine to include the Turbine libraries.
2. Once the application is created, add `@EnableTurbine` to the main Spring Boot Application class. In this example, both Turbine and Hystrix Dashboard are configured to be run on the same Spring Boot application. This is possible by adding the following annotations to the newly created Turbine application:

```
@EnableTurbine
@EnableHystrixDashboard
@SpringBootApplication
public class TurbineServerApplication {
```

3. Add the following configuration to the `.yaml` or property file to point to the instances that we are interested in monitoring:

```
spring:
 application:
 name : turbineserver
turbine:
 clusterNameExpression: new String('default')
 appConfig : search-service,search-apigateway
server:
 port: 9090
eureka:
 client:
 serviceUrl:
 defaultZone: http://localhost:8761/eureka/
```

4. The preceding configuration instructs the Turbine server to look up the Eureka server to resolve the `search-service` and `search-apigateway` services. The `search-service` and `search-apigateways` service IDs are used to register services with Eureka. Turbine uses these names to resolve the actual service host and port by checking with the Eureka server. It will then use this information to read `/hystrix.stream` from each of these instances. Turbine will then read all the individual Hystrix streams, aggregate all of them, and expose them under the Turbine server's `/turbine.stream` URL.
5. The cluster name expression is pointing to the default cluster as there is no explicit cluster configuration done in this example. If the clusters are manually configured, then the following configuration has to be use

```
turbine:
 aggregator:
 clusterConfig: [comma separated clusternames]
```

6. Change the Search service's SearchComponent to add another circuit breaker, as follows:

```
@HystrixCommand(fallbackMethod = "searchFallback")
public List<Flight> search(SearchQuery query){
```

7. Also, add @EnableCircuitBreaker to the main Application class in the Search service.
8. Add the following configuration to bootstrap.properties of the Search service. This is required because all the services are running on the same host:

```
Eureka.instance.hostname: localdomain1
```

9. Similarly, add the following in bootstrap.properties of the Search API Gateway service. This is to make sure that both the services use different hostnames:

```
eureka.instance.hostname: localdomain2
```

10. In this example, we will run two instances of search-apigateway: one on localdomain1:8095 and another one on localdomain2:8096. We will also run one instance of search-service on localdomain1:8090.
11. Run the microservices with command-line overrides to manage different host addresses, as follows:

```
java -jar -Dserver.port=8096 -Deureka.instance.
hostname=localdomain2 -Dserver.address=localdomain2 target/
chapter7.search-apigateway-1.0.jar
```

```
java -jar -Dserver.port=8095 -Deureka.instance.
hostname=localdomain1 -Dserver.address=localdomain1 target/
chapter7.search-apigateway-1.0.jar
```

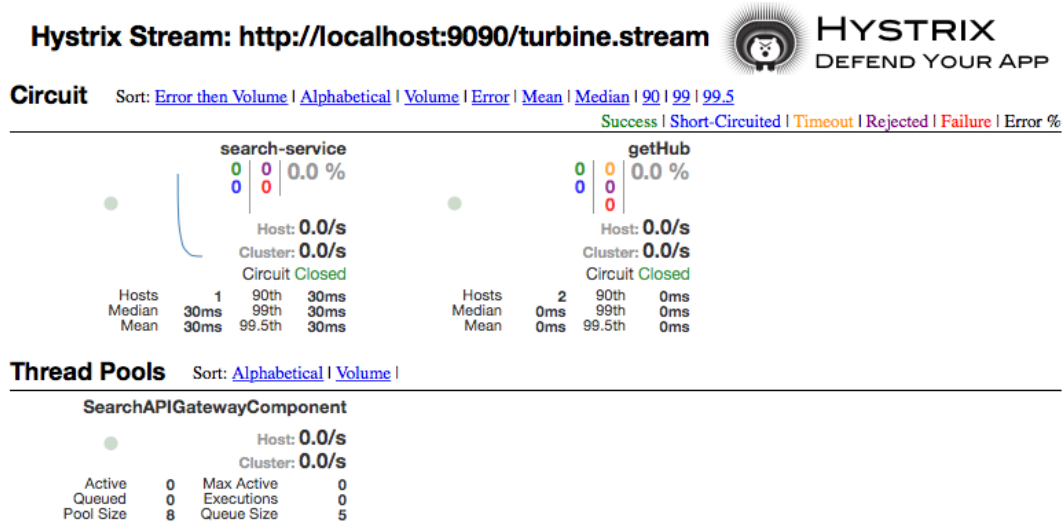
```
java -jar -Dserver.port=8090 -Deureka.instance.
hostname=localdomain1 -Dserver.address=localdomain1 target/
chapter7.search-1.0.jar
```

12. Open Hystrix Dashboard by pointing the browser to `http://localhost:9090/hystrix`.
13. Instead of giving `/hystrix.stream`, this time, we will point to `/turbine.stream`. In this example, the Turbine stream is running on 9090. Hence, the URL to be given in the Hystrix Dashboard is `http://localhost:9090/turbine.stream`.
14. Fire a few transactions by opening a browser window and hitting the following two URLs: `http://localhost:8095/hubongw` and `http://localhost:8096/hubongw`.

Once this is done, the dashboard page will show the **getHub** service.

15. Run `chapter7.website`. Execute the search transaction using the website `http://localhost:8001`.

After executing the preceding search, the dashboard page will show **search-service** as well. This is shown in the following screenshot:



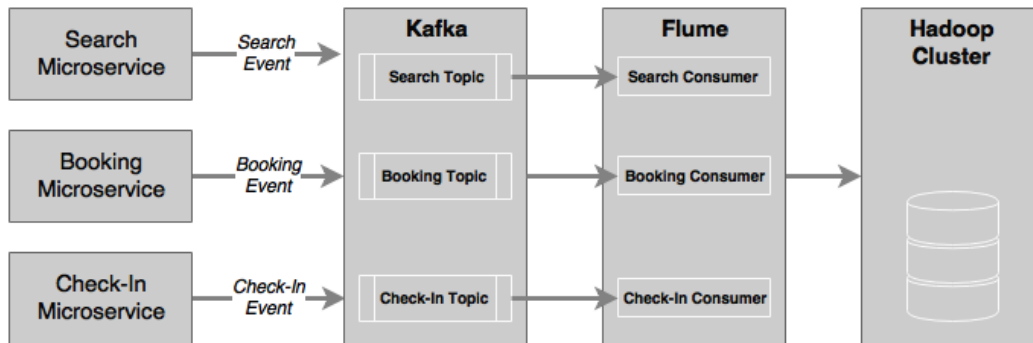
As we can see in the dashboard, **search-service** is coming from the Search microservice, and **getHub** is coming from Search API Gateway. As we have two instances of Search API Gateway, **getHub** is coming from two hosts, indicated by **Hosts 2**.

## Data analysis using data lakes

Similarly to the scenario of fragmented logs and monitoring, fragmented data is another challenge in the microservice architecture. Fragmented data poses challenges in data analytics. This data may be used for simple business event monitoring, data auditing, or even deriving business intelligence out of the data.

A data lake or data hub is an ideal solution to handling such scenarios. An event-sourced architecture pattern is generally used to share the state and state changes as events with an external data store. When there is a state change, microservices publish the state change as events. Interested parties may subscribe to these events and process them based on their requirements. A central event store may also subscribe to these events and store them in a big data store for further analysis.

One of the commonly followed architectures for such data handling is shown in the following diagram:



State change events generated from the microservice—in our case, the **Search**, **Booking**, and **Check-In** events—are pushed to a distributed high-performance messaging system, such as Kafka. A data ingestion service, such as Flume, can subscribe to these events and update them to an HDFS cluster. In some cases, these messages will be processed in real time by Spark Streaming. To handle heterogeneous sources of events, Flume can also be used between event sources and Kafka.

Spring Cloud Streams, Spring Cloud Streams modules, and Spring Data Flow are also useful as alternatives for high-velocity data ingestion.

## Summary

In this chapter, you learned about the challenges around logging and monitoring when dealing with Internet-scale microservices.

We explored the various solutions for centralized logging. You also learned about how to implement a custom centralized logging using Elasticsearch, Logstash, and Kibana (ELK). In order to understand distributed tracing, we upgraded BrownField microservices using Spring Cloud Sleuth.

In the second half of this chapter, we went deeper into the capabilities required for microservices monitoring solutions and different approaches to monitoring. Subsequently, we examined a number of tools available for microservices monitoring.



The BrownField microservices are further enhanced with Spring Cloud Hystrix and Turbine to monitor latencies and failures in inter-service communications. The examples also demonstrated how to use the circuit breaker pattern to fall back to another service in case of failures.

Finally, we also touched upon the importance of data lakes and how to integrate a data lake architecture in a microservice context.

Microservice management is another important challenge we need to tackle when dealing with large-scale microservice deployments. The next chapter will explore how containers can help in simplifying microservice management.

# 8

## Containerizing Microservices with Docker

In the context of microservices, containerized deployment is the icing on the cake. It helps microservices be more autonomous by self-containing the underlying infrastructure, thereby making the microservices cloud neutral.

This chapter will introduce the concepts and relevance of virtual machine images and the containerized deployment of microservices. Then, this chapter will further familiarize readers with building Docker images for the BrownField PSS microservices developed with Spring Boot and Spring Cloud. Finally, this chapter will also touch base on how to manage, maintain, and deploy Docker images in a production-like environment.

By the end of this chapter, you will learn about:

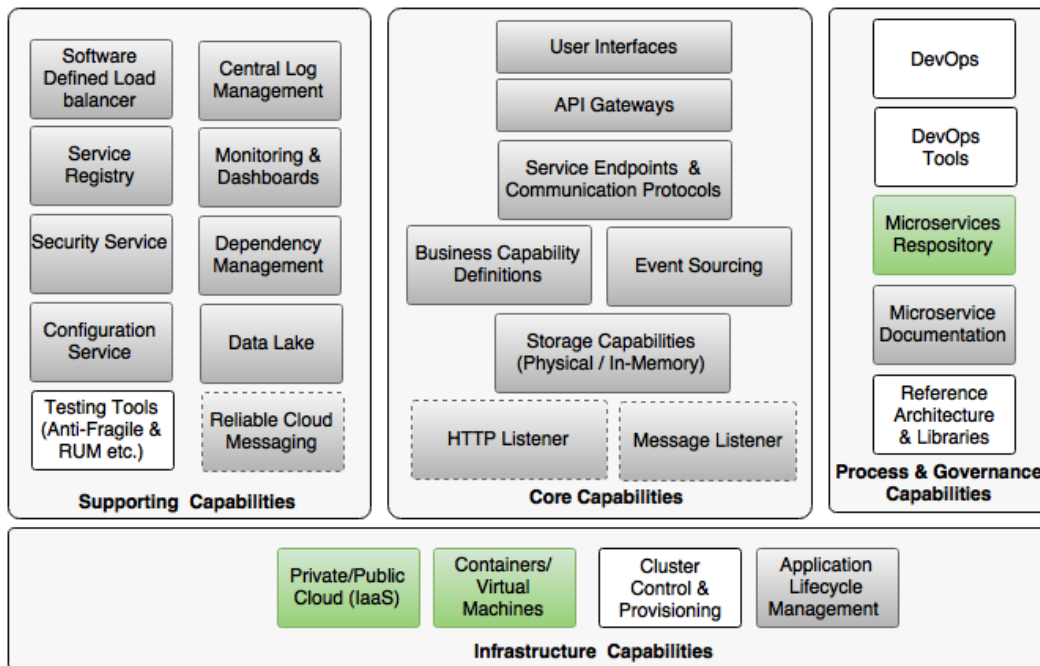
- The concept of containerization and its relevance in the context of microservices
- Building and deploying microservices as Docker images and containers
- Using AWS as an example of cloud-based Docker deployments

# Reviewing the microservice capability model

In this chapter, we will explore the following microservice capabilities from the microservice capability model discussed in *Chapter 3, Applying Microservices Concepts*:

- Containers and virtual machines
- The private/public cloud
- The microservices repository

The model is shown in the following diagram:



## Understanding the gaps in BrownField PSS microservices

In *Chapter 5, Scaling Microservices with Spring Cloud*, BrownField PSS microservices were developed using Spring Boot and Spring Cloud. These microservices are deployed as versioned fat JAR files on bare metals, specifically on a local development machine.

In *Chapter 6, Autoscaling Microservices*, the autoscaling capability was added with the help of a custom life cycle manager. In *Chapter 7, Logging and Monitoring Microservices*, challenges around logging and monitoring were addressed using centralized logging and monitoring solutions.

There are still a few gaps in our BrownField PSS implementation. So far, the implementation has not used any cloud infrastructure. Dedicated machines, as in traditional monolithic application deployments, are not the best solution for deploying microservices. Automation such as automatic provisioning, the ability to scale on demand, self-service, and payment based on usage are essential capabilities required to manage large-scale microservice deployments efficiently. In general, a cloud infrastructure provides all these essential capabilities. Therefore, a private or public cloud with the capabilities mentioned earlier is better suited to deploying Internet-scale microservices.

Also, running one microservice instance per bare metal is not cost effective. Therefore, in most cases, enterprises end up deploying multiple microservices on a single bare metal server. Running multiple microservices on a single bare metal could lead to a "noisy neighbor" problem. There is no isolation between the microservice instances running on the same machine. As a result, services deployed on a single machine may eat up others' space, thus impacting their performance.

An alternate approach is to run the microservices on VMs. However, VMs are heavyweight in nature. Therefore, running many smaller VMs on a physical machine is not resource efficient. This generally results in resource wastage. In the case of sharing a VM to deploy multiple services, we would end up facing the same issues of sharing the bare metal, as explained earlier.

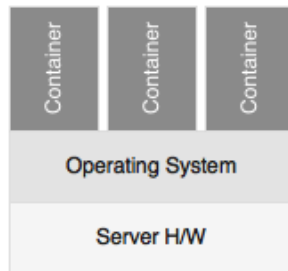
In the case of Java-based microservices, sharing a VM or bare metal to deploy multiple microservices also results in sharing JRE among microservices. This is because the fat JARs created in our BrownField PSS abstract only application code and its dependencies but not JREs. Any update on JRE installed on the machine will have an impact on all the microservices deployed on this machine. Similarly, if there are OS-level parameters, libraries, or tunings that are required for specific microservices, then it will be hard to manage them on a shared environment.

One microservice principle insists that it should be self-contained and autonomous by fully encapsulating its end-to-end runtime environment. In order to align with this principle, all components, such as the OS, JRE, and microservice binaries, have to be self-contained and isolated. The only option to achieve this is to follow the approach of deploying one microservice per VM. However, this will result in underutilized virtual machines, and in many cases, extra cost due to this can nullify benefits of microservices

# What are containers?

Containers are not revolutionary, ground-breaking concepts. They have been in action for quite a while. However, the world is witnessing the re-entry of containers, mainly due to the wide adoption of cloud computing. The shortcomings of traditional virtual machines in the cloud computing space also accelerated the use of containers. Container providers such as **Docker** simplified container technologies to a great extent which also enabled a large adoption of container technologies in today's world. The recent popularity of DevOps and microservices also acted as a catalyst for the rebirth of container technologies.

So, what are containers? Containers provide private spaces on top of the operating system. This technique is also called operating system virtualization. In this approach, the kernel of the operating system provides isolated virtual spaces. Each of these virtual spaces is called a container or **virtual engine (VE)**. Containers allow processes to run on an isolated environment on top of the host operating system. A representation of multiple containers running on the same host is shown as follows:



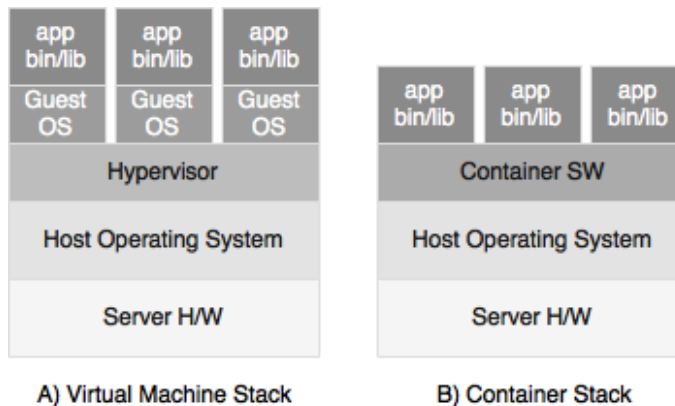
Containers are easy mechanisms to build, ship, and run compartmentalized software components. Generally, containers package all the binaries and libraries that are essential for running an application. Containers reserve their own filesystem, IP address, network interfaces, internal processes, namespaces, OS libraries, application binaries, dependencies, and other application configurations

There are billions of containers used by organizations. Moreover, there are many large organizations heavily investing in container technologies. Docker is far ahead of the competition, supported by many large operating system vendors and cloud providers. **Lmctfy**, **SystemdNspawn**, **Rocket**, **Drawbridge**, **LXD**, **Kurma**, and **Calico** are some of the other containerization solutions. Open container specification is also under development.

# The difference between VMs and containers

VMs such as **Hyper-V**, **VMWare**, and **Zen** were popular choices for data center virtualization a few years ago. Enterprises experienced a cost saving by implementing virtualization over the traditional bare metal usage. It has also helped many enterprises utilize their existing infrastructure in a much more optimized manner. As VMs support automation, many enterprises experienced that they had to make lesser management effort with virtual machines. Virtual machines also helped organizations get isolated environments for applications to run in.

Prima facie, both virtualization and containerization exhibit exactly the same characteristics. However, in a nutshell, containers and virtual machines are not the same. Therefore, it is unfair to make an apple-to-apple comparison between VMs and containers. Virtual machines and containers are two different techniques and address different problems of virtualization. This difference is evident from the following diagram:



Virtual machines operate at a much lower level compared to containers. VMs provide hardware virtualization, such as that of CPUs, motherboards, memory, and so on. A VM is an isolated unit with an embedded operating system, generally called a **Guest OS**. VMs replicate the whole operating system and run it within the VM with no dependency on the host operating system environment. As VMs embed the full operating system environment, these are heavyweight in nature. This is an advantage as well as a disadvantage. The advantage is that VMs offer complete isolation to the processes running on VMs. The disadvantage is that it limits the number of VMs one can spin up in a bare metal due to the resource requirements of VMs.

The size of a VM has a direct impact on the time to start and stop it. As starting a VM in turn boots the OS, the start time for VMs is generally high. VMs are more friendly with infrastructure teams as it requires a low level of infrastructure competency to manage VMs.

In the container world, containers do not emulate the entire hardware or operating system. Unlike VMs, containers share certain parts of the host kernel and operating system. There is no concept of guest OS in the case of containers. Containers provide an isolated execution environment directly on top of the host operating system. This is its advantage as well as disadvantage. The advantage is that it is lighter as well as faster. As containers on the same machine share the host operating system, the overall resource utilization of containers is fairly small. As a result, many smaller containers can be run on the same machine, as compared to heavyweight VMs. As containers on the same host share the host operating system, there are limitations as well. For example, it is not possible to set iptables firewall rules inside a container. Processes inside the container are completely independent from the processes on different containers running on the same host.

Unlike VMs, container images are publically available on community portals. This makes developers' lives much easier as they don't have to build the images from scratch; instead, they can now take a base image from certified sources and add additional layers of software components on top of the downloaded base image.

The lightweight nature of the containers is also opening up a plethora of opportunities, such as automated build, publishing, downloading, copying, and so on. The ability to download, build, ship, and run containers with a few commands or to use REST APIs makes containers more developer friendly. Building a new container does not take more than a few seconds. Containers are now part and parcel of continuous delivery pipelines as well.

In summary, containers have many advantages over VMs, but VMs have their own exclusive strengths. Many organizations use both containers and VMs, such as by running containers on top of VMs.

# The benefits of containers

We have already considered the many benefits of containers over VMs. This section will explain the overall benefits of containers beyond the benefits of VM

- **Self-contained:** Containers package the essential application binaries and their dependencies together to make sure that there is no disparity between different environments such as development, testing, or production. This promotes the concept of Twelve-Factor applications and that of immutable containers. Spring Boot microservices bundle all the required application dependencies. Containers stretch this boundary further by embedding JRE and other operating system-level libraries, configurations, and so on, if there are any.
- **Lightweight:** Containers, in general, are smaller in size with a lighter footprint. The smallest container, Alpine, has a size of less than 5 MB. The simplest Spring Boot microservice packaged with an Alpine container with Java 8 would only come to around 170 MB in size. Though the size is still on the higher side, it is much less than the VM image size, which is generally in GBs. The smaller footprint of containers not only helps spin new containers quickly but also makes building, shipping, and storing easier.
- **Scalable:** As container images are smaller in size and there is no OS booting at startup, containers are generally faster to spin up and shut down. This makes containers the popular choice for cloud-friendly elastic applications.
- **Portable:** Containers provide portability across machines and cloud providers. Once the containers are built with all the dependencies, they can be ported across multiple machines or across multiple cloud providers without relying on the underlying machines. Containers are portable from desktops to different cloud environments.
- **Lower license cost:** Many software license terms are based on the physical core. As containers share the operating system and are not virtualized at the physical resources level, there is an advantage in terms of the license cost.
- **DevOps:** The lightweight footprint of containers makes it easy to automate builds and publish and download containers from remote repositories. This makes it easy to use in Agile and DevOps environments by integrating with automated delivery pipelines. Containers also support the concept of *build once* by creating immutable containers at build time and moving them across multiple environments. As containers are not deep into the infrastructure, multidisciplinary DevOps teams can manage containers as part of their day-to-day life.
- **Version controlled:** Containers support versions by default. This helps build versioned artifacts, just as with versioned archive files.



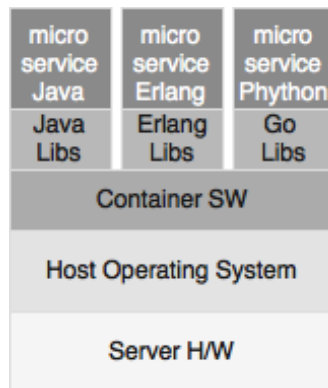
- **Reusable:** Container images are reusable artifacts. If an image is built by assembling a number of libraries for a purpose, it can be reused in similar situations.
- **Immutable containers:** In this concept, containers are created and disposed of after usage. They are never updated or patched. Immutable containers are used in many environments to avoid complexities in patching deployment units. Patching results in a lack of traceability and an inability to recreate environments consistently.

## Microservices and containers

There is no direct relationship between microservices and containers. Microservices can run without containers, and containers can run monolithic applications. However, there is a sweet spot between microservices and containers.

Containers are good for monolithic applications, but the complexities and the size of the monolith application may kill some of the benefits of the containers. For example, spinning new containers quickly may not be easy with monolithic applications. In addition to this, monolithic applications generally have local environment dependencies, such as the local disk, stovepipe dependencies with other systems, and so on. Such applications are difficult to manage with container technologies. This is where microservices go hand in hand with containers.

The following diagram shows three polyglot microservices running on the same host machine and sharing the same operating system but abstracting the runtime environment:



The real advantage of containers can be seen when managing many polyglot microservices—for instance, one microservice in Java and another one in Erlang or some other language. Containers help developers package microservices written in any language or technology in a platform- and technology-agnostic fashion and uniformly distribute them across multiple environments. Containers eliminate the need to have different deployment management tools to handle polyglot microservices. Containers not only abstract the execution environment but also how to access the services. Irrespective of the technologies used, containerized microservices expose REST APIs. Once the container is up and running, it binds to certain ports and exposes its APIs. As containers are self-contained and provide full stack isolation among services, in a single VM or bare metal, one can run multiple heterogeneous microservices and handle them in a uniform way.

## Introduction to Docker

The previous sections talked about containers and their benefits. Containers have been in the business for years, but the popularity of Docker has given containers a new outlook. As a result, many container definitions and perspectives emerged from the Docker architecture. Docker is so popular that even containerization is referred to as **dockerization**.

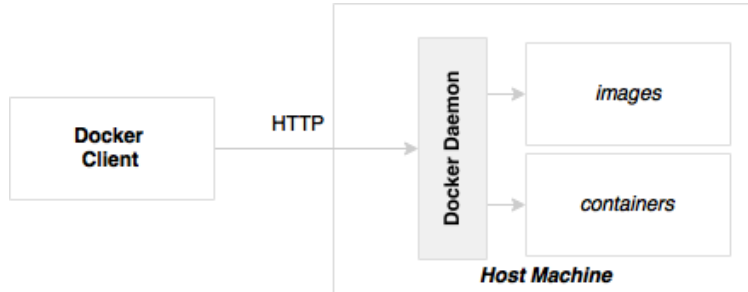
Docker is a platform to build, ship, and run lightweight containers based on Linux kernels. Docker has default support for Linux platforms. It also has support for Mac and Windows using **Boot2Docker**, which runs on top of Virtual Box.

Amazon **EC2 Container Service (ECS)** has out-of-the-box support for Docker on AWS EC2 instances. Docker can be installed on bare metals and also on traditional virtual machines such as VMWare or Hyper-V.

## The key components of Docker

A Docker installation has two key components: a **Docker daemon** and a **Docker client**. Both the Docker daemon and Docker client are distributed as a single binary.

The following diagram shows the key components of a Docker installation:



### The Docker daemon

The Docker daemon is a server-side component that runs on the host machine responsible for building, running, and distributing Docker containers. The Docker daemon exposes APIs for the Docker client to interact with the daemon. These APIs are primarily REST-based endpoints. One can imagine that the Docker daemon as a controller service running on the host machine. Developers can programmatically use these APIs to build custom clients as well.

### The Docker client

The Docker client is a remote command-line program that interacts with the Docker daemon through either a socket or REST APIs. The CLI can run on the same host as the daemon is running on or it can run on a completely different host and connect to the daemon remotely. Docker users use the CLI to build, ship, and run Docker containers.

# Docker concepts

The Docker architecture is built around a few concepts: images, containers, the registry, and the Dockerfile

## Docker images

One of the key concepts of Docker is the image. A Docker image is the read-only copy of the operating system libraries, the application, and its libraries. Once an image is created, it is guaranteed to run on any Docker platform without alterations.

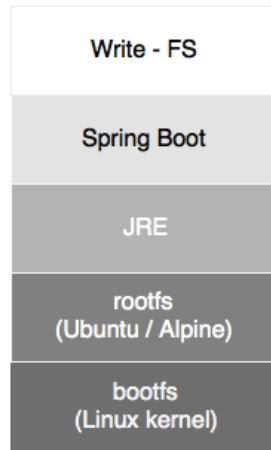
In Spring Boot microservices, a Docker image packages operating systems such as Ubuntu, Alpine, JRE, and the Spring Boot fat application JAR file. It also includes instructions to run the application and expose the services:



As shown in the diagram, Docker images are based on a layered architecture in which the base image is one of the flavors of Linux. Each layer, as shown in the preceding diagram, gets added to the base image layer with the previous image as the parent layer. Docker uses the concept of a union filesystem to combine all these layers into a single image, forming a single filesystem

In typical cases, developers do not build Docker images from scratch. Images of an operating system, or other common libraries, such as Java 8 images, are publicly available from trusted sources. Developers can start building on top of these base images. The base image in Spring microservices can be JRE 8 rather than starting from a Linux distribution image such as Ubuntu.

Every time we rebuild the application, only the changed layer gets rebuilt, and the remaining layers are kept intact. All the intermediate layers are cached, and hence, if there is no change, Docker uses the previously cached layer and builds it on top. Multiple containers running on the same machine with the same type of base images would reuse the base image, thus reducing the size of the deployment. For instance, in a host, if there are multiple containers running with Ubuntu as the base image, they all reuse the same base image. This is applicable when publishing or downloading images as well:



As shown in the diagram, the first layer in the image is a boot filesystem called `bootfs`, which is similar to the Linux kernel and the boot loader. The boot filesystem acts as a virtual filesystem for all images.

On top of the boot filesystem, the operating system filesystem is placed, which is called `rootfs`. The root filesystem adds the typical operating system directory structure to the container. Unlike in the Linux systems, `rootfs`, in the case of Docker, is on a read-only mode.

On top of `rootfs`, other required images are placed as per the requirements. In our case, these are JRE and the Spring Boot microservice JARs. When a container is initiated, a writable filesystem is placed on top of all the other filesystems for the processes to run. Any changes made by the process to the underlying filesystem are not reflected in the actual container. Instead, these are written to the writable filesystem. This writable filesystem is volatile. Hence, the data is lost once the container is stopped. Due to this reason, Docker containers are ephemeral in nature.

The base operating system packaged inside Docker is generally a minimal copy of just the OS filesystem. In reality the process running on top may not use the entire OS services. In a Spring Boot microservice, in many cases, the container just initiates a CMD and JVM and then invokes the Spring Boot fat JAR.

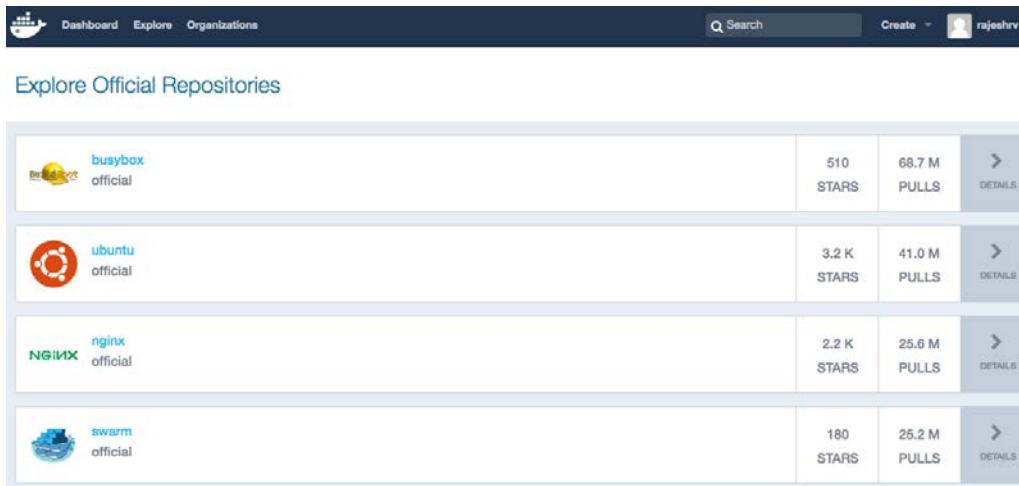
## Docker containers

Docker containers are the running instances of a Docker image. Containers use the kernel of the host operating system when running. Hence, they share the host kernel with other containers running on the same host. The Docker runtime ensures that the container processes are allocated with their own isolated process space using kernel features such as **cgroups** and the kernel **namespace** of the operating system. In addition to the resource fencing, containers get their own filesystem and network configurations as well

The containers, when instantiated, can have specific resource allocations, such as the memory and CPU. Containers, when initiated from the same image, can have different resource allocations. The Docker container, by default, gets an isolated **subnet** and **gateway** to the network. The network has three modes.

## The Docker registry

The Docker registry is a central place where Docker images are published and downloaded from. The URL <https://hub.docker.com> is the central registry provided by Docker. The Docker registry has public images that one can download and use as the base registry. Docker also has private images that are specific to the accounts created in the Docker registry. The Docker registry screenshot is shown as follows:



The screenshot shows the Docker Hub 'Explore Official Repositories' page. It features a dark blue header with navigation links (Dashboard, Explore, Organizations), a search bar, and a user profile for 'rajeshrv'. Below the header, a table lists four official Docker images: busybox, ubuntu, nginx, and swarm. Each row includes the image icon, name, version, star count, pull count, and a details link.

Image	Stars	Pulls	Details
busybox official	510 STARS	68.7 M PULLS	> DETAILS
ubuntu official	3.2 K STARS	41.0 M PULLS	> DETAILS
nginx official	2.2 K STARS	25.6 M PULLS	> DETAILS
swarm official	180 STARS	25.2 M PULLS	> DETAILS

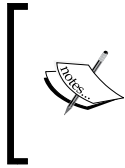
Docker also offers **Docker Trusted Registry**, which can be used to set up registries locally on premises.

## Dockerfile

A Dockerfile is a build or scripting file that contains instructions to build a Docker image. There can be multiple steps documented in the Dockerfile, starting from getting a base image. A Dockerfile is a text file that is generally named `Dockerfile`. The `docker build` command looks up Dockerfile for instructions to build. One can compare a Dockerfile to a `pom.xml` file used in a Maven build.

## Deploying microservices in Docker

This section will operationalize our learning by showcasing how to build containers for our BrownField PSS microservices.



The full source code of this chapter is available under the Chapter 8 project in the code files. Copy `chapter7.configserver`, `chapter7.eurekaclient`, `chapter7.search`, `chapter7.search-apigateway`, and `chapter7.website` into a new STS workspace and rename them `chapter8.*`.

Perform the following steps to build Docker containers for BrownField PSS microservices:

1. Install Docker from the official Docker site at <https://www.docker.com>. Follow the **Get Started** link for the download and installation instructions based on the operating system of choice. Once installed, use the following command to verify the installation:  

```
$docker -version
```

```
Docker version 1.10.1, build 9e83765
```
2. In this section, we will take a look at how to dockerize the **Search** (`chapter8.search`) microservice, the **Search API Gateway** (`chapter8.search-apigateway`) microservice, and the **Website** (`chapter8.website`) Spring Boot application.
3. Before we make any changes, we need to edit `bootstrap.properties` to change the config server URL from `localhost` to the IP address as `localhost` is not resolvable from within the Docker containers. In the real world, this will point to a DNS or load balancer, as follows:  

```
spring.cloud.config.uri=http://192.168.0.105:8888
```



Replace the IP address with the IP address of your machine.

4. Similarly, edit `search-service.properties` on the Git repository and change `localhost` to the IP address. This is applicable for the Eureka URL as well as the RabbitMQ URL. Commit back to Git after updating. You can do this via the following code:

```
spring.application.name=search-service
spring.rabbitmq.host=192.168.0.105
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
originairports.shutdown:JFK
eureka.client.serviceUrl.defaultZone: http://192.168.0.105:8761/
eureka/
spring.cloud.stream.bindings.inventoryQ=inventoryQ
```

5. Change the RabbitMQ configuration file `rabbitmq.config` by uncommenting the following line to provide access to guest. By default, guest is restricted to be accessed from localhost only:

```
{loopback_users, []}
```

The location of `rabbitmq.config` will be different for different operating systems.

6. Create a Dockerfile under the root directory of the Search microservice, as follows:

```
FROM frovlad/alpine-oraclejdk8
VOLUME /tmp
ADD target/search-1.0.jar search.jar
EXPOSE 8090
ENTRYPOINT ["java","-jar","/search.jar"]
```

The following is a quick examination of the contents of the Dockerfile:

- `FROM frovlad/alpine-oraclejdk8`: This tells the Docker build to use a specific `alpine-oraclejdk8` version as the basic image for this build. The `frovlad` indicates the repository to locate the `alpine-oraclejdk8` image. In this case, it is an image built with Alpine Linux and Oracle JDK 8. This will help layer our application on top of the base image without setting up Java libraries ourselves. In this case, as this image is not available on our local image store, the Docker build will go ahead and download this image from the remote Docker Hub registry.



- `VOLUME /tmp`: This enables access from the container to the directory specified in the host machine. In our case, this points to the `tmp` directory in which the Spring Boot application creates working directories for Tomcat. The `tmp` directory is a logical one for the container, which indirectly points to one of the local directories of the host.
  - `ADD target/search-1.0.jar search.jar`: This adds the application binary file to the container with the destination filename specified. In this case, the Docker build copies `target/search-1.0.jar` to the container as `search.jar`.
  - `EXPOSE 8090`: This is to tell the container how to do port mapping. This associates 8090 with external port binding for the internal Spring Boot service.
  - `ENTRYPOINT ["java", "-jar", "/search.jar"]`: This tells the container which default application to run when a container is started. In this case, we are pointing to the Java process and the Spring Boot fat JAR file to initiate the service.
7. The next step is to run `docker build` from the folder in which the Dockerfile is stored. This will download the base image and run the entries in the Dockerfile one after the other, as follows:

```
docker build -t search:1.0 .
```

The output of this command will be as follows:

```
rvslab:chapter8.search rajeshrv$ docker build -t search:1.0 .
Sending build context to Docker daemon 48.34 MB
Step 1 : FROM frovlad/alpine-oraclejdk8
--> 5b8d90632c89
Step 2 : VOLUME /tmp
--> Using cache
--> c79a1b3275d4
Step 3 : ADD target/search-1.0.jar app.jar
--> 7766e630f139
Removing intermediate container f2ac976e781d
Step 4 : EXPOSE 8090
--> Running in 730300fa66a9
--> e058cc1615da
Removing intermediate container 730300fa66a9
Step 5 : ENTRYPOINT java -jar app.jar
--> Running in b79116f3e54b
--> 5a8d0d6e0bf7
Removing intermediate container b79116f3e54b
Successfully built 5a8d0d6e0bf7
rvslab:chapter8.search rajeshrv$ █
```

8. Repeat the same steps for Search API Gateway and Website.
9. Once the images are created, they can be verified by typing the following command. This command will list out the images and their details, including the size of image files

```
docker images
```

The output will be as follows:

```
rvslab:chapter8 rajeshrv$ docker images
REPOSITORY TAG IMAGE ID
website 1.0 263605f253f3
search-apigateway 1.0 322890ae3ec1
search 1.0 f094b72d1b41
```

10. The next thing to do is run the Docker container. This can be done with the `docker run` command. This command will load and run the container. On starting, the container calls the Spring Boot executable JAR to start the microservice.

Before starting the containers, ensure that the Config and the Eureka servers are running:

```
docker run --net host -p 8090:8090 -t search:1.0
docker run --net host -p 8095:8095 -t search-apigateway:1.0
docker run --net host -p 8001:8001 -t website:1.0
```

The preceding command starts the Search and Search API Gateway microservices and Website.

In this example, we are using the host network (`--net host`) instead of the bridge network to avoid Eureka registering with the Docker container name. This can be corrected by overriding `EurekaInstanceConfigBean`. The host option is less isolated compared to the bridge option from the network perspective. The advantage and disadvantage of host versus bridge depends on the project.

11. Once all the services are fully started, verify with the `docker ps` command, as shown in the following screenshot:

```
rvslab:chapter8 rajeshrv$ docker ps
CONTAINER ID IMAGE COMMAND
32af53b56945 website:1.0 "java -jar /website.j"
ce35355aea65 search-apigateway:1.0 "java -jar /search-ap"
5577c5107fc6 search:1.0 "java -jar /search.ja"
4a423d0872b5 registry:2 "/bin/registry /etc/d"
rvslab:chapter8 rajeshrv$
```

12. The next step is to point the browser to `http://192.168.99.100:8001`. This will open the BrownField PSS website.

Note the IP address. This is the IP address of the Docker machine if you are running with Boot2Docker on Mac or Windows. In Mac or Windows, if the IP address is not known, then type the following command to find out the Docker machine's IP address for the default machine:

```
docker-machine ip default
```

If Docker is running on Linux, then this is the host IP address.

Apply the same changes to **Booking**, **Fares**, **Check-in**, and their respective gateway microservices.

## Running RabbitMQ on Docker

As our example also uses RabbitMQ, let's explore how to set up RabbitMQ as a Docker container. The following command pulls the RabbitMQ image from Docker Hub and starts RabbitMQ:

```
docker run -net host rabbitmq3
```

Ensure that the URL in `*-service.properties` is changed to the Docker host's IP address. Apply the earlier rule to find out the IP address in the case of Mac or Windows.

## Using the Docker registry

The Docker Hub provides a central location to store all the Docker images. The images can be stored as public as well as private. In many cases, organizations deploy their own private registries on premises due to security-related concerns.

Perform the following steps to set up and run a local registry:

1. The following command will start a registry, which will bind the registry on port 5000:

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

2. Tag `search:1.0` to the registry, as follows:

```
docker tag search:1.0 localhost:5000/search:1.0
```

3. Then, push the image to the registry via the following command:

```
docker push localhost:5000/search:1.0
```

4. Pull the image back from the registry, as follows:

```
docker pull localhost:5000/search:1.0
```

## Setting up the Docker Hub

In the previous chapter, we played with a local Docker registry. This section will show how to set up and use the Docker Hub to publish the Docker containers. This is a convenient mechanism to globally access Docker images. Later in this chapter, Docker images will be published to the Docker Hub from the local machine and downloaded from the EC2 instances.

In order to do this, create a public Docker Hub account and a repository.

For Mac, follow the steps as per the following URL: [https://docs.docker.com/mac/step\\_five/](https://docs.docker.com/mac/step_five/).

In this example, the Docker Hub account is created using the `brownfield` username.

The registry, in this case, acts as the microservices repository in which all the dockerized microservices will be stored and accessed. This is one of the capabilities explained in the microservices capability model.

## Publishing microservices to the Docker Hub

In order to push dockerized services to the Docker Hub, follow these steps. The first command tags the Docker image, and the second one pushes the Docker image to the Docker Hub repository:

```
docker tag search:1.0brownfield/search:1.0
```

```
docker push brownfield/search:1.0
```

To verify whether the container images are published, go to the Docker Hub repository at <https://hub.docker.com/u/brownfield>.

Repeat this step for all the other BrownField microservices as well. At the end of this step, all the services will be published to the Docker Hub.

## Microservices on the cloud

One of the capabilities mentioned in the microservices capability model is the use of the cloud infrastructure for microservices. Earlier in this chapter, we also explored the necessity of using the cloud for microservices deployments. So far, we have not deployed anything to the cloud. As we have eight microservices in total—Config-server, Eureka-server, Turbine, RabbitMQ, Elasticsearch, Kibana, and Logstash—in our overall BrownField PSS microservices ecosystem, it is hard to run all of them on the local machine.

In the rest of this book, we will operate using AWS as the cloud platform to deploy BrownField PSS microservices.

## Installing Docker on AWS EC2

In this section, we will install Docker on the EC2 instance.

This example assumes that readers are familiar with AWS and an account is already created on AWS.

Perform the following steps to set up Docker on EC2:

1. Launch a new EC2 instance. In this case, if we have to run all the instances together, we may need a large instance. The example uses **t2.large**.  
In this example, the following Ubuntu AMI image is used: `ubuntu-trusty-14.04-amd64-server-20160114.5 (ami-fce3c696)`.
2. Connect to the EC2 instance and run the following commands:  

```
sudo apt-get update
sudo apt-get install docker.io
```
3. The preceding command will install Docker on an EC2 instance. Verify the installation with the following command:  

```
docker version
```

## Running BrownField services on EC2

In this section, we will set up BrownField microservices on the EC2 instances created. In this case, the build is set up in the local desktop machine, and the binaries will be deployed to AWS.

Perform the following steps to set up services on an EC2 instance:

1. Install Git via the following command:  
`sudo apt-get install git`
2. Create a Git repository on any folder of your choice.
3. Change the Config server's `bootstrap.properties` to point to the appropriate Git repository created for this example.
4. Change the `bootstrap.properties` of all the microservices to point to the config-server using the private IP address of the EC2 instance
5. Copy all `*.properties` from the local Git repository to the EC2 Git repository and perform a commit.
6. Change the Eureka server URLs and RabbitMQ URLs in the `*.properties` file to match the EC2 private IP address. Commit the changes to Git once they have been completed.
7. On the local machine, recompile all the projects and create Docker images for the `search`, `search-apigateway`, and `website` microservices. Push all of them to the Docker Hub registry.
8. Copy the config-server and the Eureka-server binaries from the local machine to the EC2 instance.
9. Set up Java 8 on the EC2 instance.
10. Then, execute the following commands in sequence:  
`java -jar config-server.jar`  
`java -jar eureka-server.jar`  
`docker run -net host rabbitmq:3`  
`docker run --net host -p 8090:8090 rajeshrv/search:1.0`  
`docker run --net host -p 8095:8095 rajeshrv/search-apigateway:1.0`  
`docker run --net host -p 8001:8001 rajeshrv/website:1.0`
11. Check whether all the services are working by opening the URL of the website and executing a search. Note that we will use the public IP address in this case: `http://54.165.128.23:8001`.

## Updating the life cycle manager

In *Chapter 6, Autoscaling Microservices*, we considered a life cycle manager to automatically start and stop instances. We used SSH and executed a Unix script to start the Spring Boot microservices on the target machine. With Docker, we no longer need SSH connections as the Docker daemon provides REST-based APIs to start and stop instances. This greatly simplifies the complexities of the deployment engine component of the life cycle manager.

In this section, we will not rewrite the life cycle manager. By and large, we will replace the life cycle manager in the next chapter.

## The future of containerization – unikernels and hardened security

Containerization is still evolving, but the number of organizations adopting containerization techniques has gone up in recent times. While many organizations are aggressively adopting Docker and other container technologies, the downside of these techniques is still in the size of the containers and security concerns.

Currently, Docker images are generally heavy. In an elastic automated environment, where containers are created and destroyed quite frequently, size is still an issue. A larger size indicates more code, and more code means that it is more prone to security vulnerabilities.

The future is definitely in small footprint containers. Docker is working on unikernels, lightweight kernels that can run Docker even on low-powered IoT devices. Unikernels are not full-fledged operating systems, but they provide the basic necessary libraries to support the deployed applications.

The security issues of containers are much discussed and debated. The key security issues are around the user namespace segregation or user ID isolation. If the container is on root, then it can by default gain the root privilege of the host. Using container images from untrusted sources is another security concern. Docker is bridging these gaps as quickly as possible, but there are many organizations that use a combination of VMs and Docker to circumvent some of the security concerns.

## Summary

In this chapter, you learned about the need to have a cloud environment when dealing with Internet-scale microservices.

We explored the concept of containers and compared them with traditional virtual machines. You also learned the basics of Docker, and we explained the concepts of Docker images, containers, and registries. The importance and benefits of containers were explained in the context of microservices.

This chapter then switched to a hands-on example by dockerizing the BrownField microservice. We demonstrated how to deploy the Spring Boot microservice developed earlier on Docker. You learned the concept of registries by exploring a local registry as well as the Docker Hub to push and pull dockerized microservices.

As the last step, we explored how to deploy a dockerized BrownField microservice in the AWS cloud environment.





# 9

## Managing Dockerized Microservices with Mesos and Marathon

In an Internet-scale microservices deployment, it is not easy to manage thousands of dockerized microservices. It is essential to have an infrastructure abstraction layer and a strong cluster control platform to successfully manage Internet-scale microservice deployments.

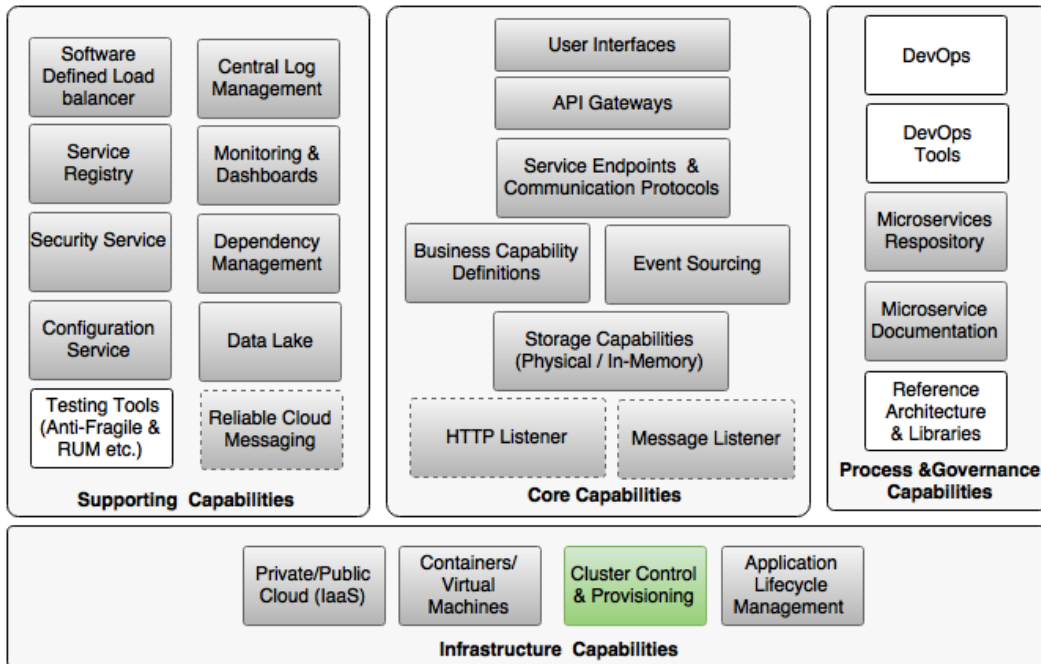
This chapter will explain the need and use of Mesos and Marathon as an infrastructure abstraction layer and a cluster control system, respectively, to achieve optimized resource usage in a cloud-like environment when deploying microservices at scale. This chapter will also provide a step-by-step approach to setting up Mesos and Marathon in a cloud environment. Finally, this chapter will demonstrate how to manage dockerized microservices in the Mesos and Marathon environment.

By the end of this chapter, you will have learned about:

- The need to have an abstraction layer and cluster control software
- Mesos and Marathon from the context of microservices
- Managing dockerized BrownField Airline's PSS microservices with Mesos and Marathon

# Reviewing the microservice capability model

In this chapter, we will explore the **Cluster Control & Provisioning** microservices capability from the microservices capability model discussed in *Chapter 3, Applying Microservices Concepts*:



## The missing pieces

In *Chapter 8, Containerizing Microservices with Docker*, we discussed how to dockerize BrownField Airline's PSS microservices. Docker helped package the JVM runtime and OS parameters along with the application so that there is no special consideration required when moving dockerized microservices from one environment to another. The REST APIs provided by Docker have simplified the life cycle manager's interaction with the target machine in starting and stopping artifacts.

In a large-scale deployment, with hundreds and thousands of Docker containers, we need to ensure that Docker containers run with their own resource constraints, such as memory, CPU, and so on. In addition to this, there may be rules set for Docker deployments, such as replicated copies of the container should not be run on the same machine. Also, a mechanism needs to be in place to optimally use the server infrastructure to avoid incurring extra cost.

There are organizations that deal with billions of containers. Managing them manually is next to impossible. In the context of large-scale Docker deployments, some of the key questions to be answered are:

- How do we manage thousands of containers?
- How do we monitor them?
- How do we apply rules and constraints when deploying artifacts?
- How do we ensure that we utilize containers properly to gain resource efficiency?
- How do we ensure that at least a certain number of minimal instances are running at any point in time?
- How do we ensure dependent services are up and running?
- How do we do rolling upgrades and graceful migrations?
- How do we roll back faulty deployments?

All these questions point to the need to have a solution to address two key capabilities, which are as follows:

- A cluster abstraction layer that provides a uniform abstraction over many physical or virtual machines
- A cluster control and init system to manage deployments intelligently on top of the cluster abstraction

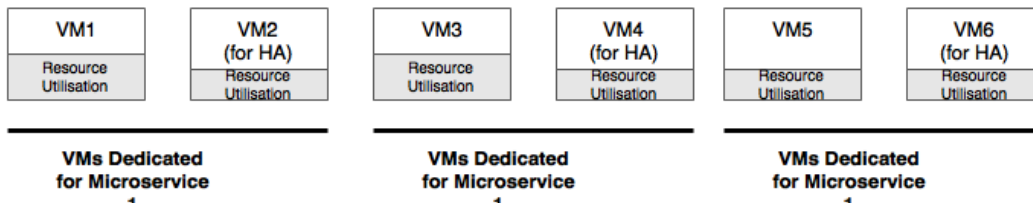
The life cycle manager is ideally placed to deal with these situations. One can add enough intelligence to the life cycle manager to solve these issues. However, before attempting to modify the life cycle manager, it is important to understand the role of cluster management solutions a bit more.

## Why cluster management is important

As microservices break applications into different micro-applications, many developers request more server nodes for deployment. In order to manage microservices properly, developers tend to deploy one microservice per VM, which further drives down the resource utilization. In many cases, this results in an overallocation of CPUs and memory.

In many deployments, the high-availability requirements of microservices force engineers to add more and more service instances for redundancy. In reality, though it provides the required high availability, this will result in underutilized server instances.

In general, microservice deployment requires more infrastructure compared to monolithic application deployments. Due to the increase in cost of the infrastructure, many organizations fail to see the value of microservices:



In order to address the issue stated before, we need a tool that is capable of the following:

- Automating a number of activities, such as the allocation of containers to the infrastructure efficiently and keeping it transparent to developers and administrators
- Providing a layer of abstraction for the developers so that they can deploy their application against a data center without knowing which machine is to be used to host their applications
- Setting rules or constraints against deployment artifacts
- Offering higher levels of agility with minimal management overheads for developers and administrators, perhaps with minimal human interaction
- Building, deploying, and managing the application's cost effectively by driving a maximum utilization of the available resources

Containers solve an important issue in this context. Any tool that we select with these capabilities can handle containers in a uniform way, irrespective of the underlying microservice technologies.

# What does cluster management do?

Typical cluster management tools help virtualize a set of machines and manage them as a single cluster. Cluster management tools also help move the workload or containers across machines while being transparent to the consumer. Technology evangelists and practitioners use different terminologies, such as cluster orchestration, cluster management, data center virtualization, container schedulers, or container life cycle management, container orchestration, data center operating system, and so on.

Many of these tools currently support both Docker-based containers as well as noncontainerized binary artifact deployments, such as a standalone Spring Boot application. The fundamental function of these cluster management tools is to abstract the actual server instance from the application developers and administrators.

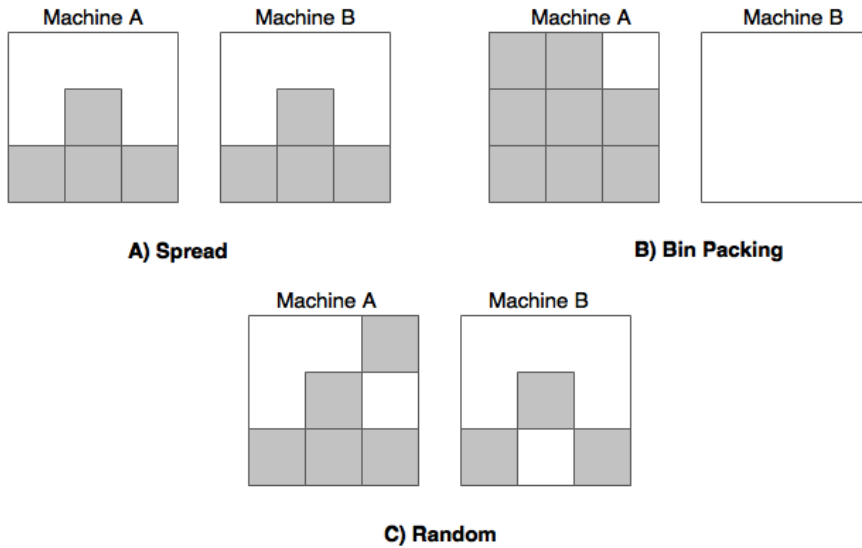
Cluster management tools help the self-service and provisioning of infrastructure rather than requesting the infrastructure teams to allocate the required machines with a predefined specification. In this automated cluster management approach machines are no longer provisioned upfront and preallocated to the applications. Some of the cluster management tools also help virtualize data centers across many heterogeneous machines or even across data centers, and create an elastic, private cloud-like infrastructure. There is no standard reference model for cluster management tools. Therefore, the capabilities vary between vendors.

Some of the key capabilities of cluster management software are summarized as follows:

- **Cluster management:** It manages a cluster of VMs and physical machines as a single large machine. These machines could be heterogeneous in terms of resource capabilities, but they are, by and large, machines with Linux as the operating system. These virtual clusters can be formed on the cloud, on-premises, or a combination of both.
- **Deployments:** It handles the automatic deployment of applications and containers with a large set of machines. It supports multiple versions of the application containers and also rolling upgrades across a large number of cluster machines. These tools are also capable of handling the rollback of faulty promotes.
- **Scalability:** It handles the automatic and manual scalability of application instances as and when required, with optimized utilization as the primary goal.
- **Health:** It manages the health of the cluster, nodes, and applications. It removes faulty machines and application instances from the cluster.

- **Infrastructure abstraction:** It abstracts the developers from the actual machine on which the applications are deployed. The developers need not worry about the machine, its capacity, and so on. It is entirely the cluster management software's decision to decide how to schedule and run the applications. These tools also abstract machine details, their capacity, utilization, and location from the developers. For application owners, these are equivalent to a single large machine with almost unlimited capacity.
- **Resource optimization:** The inherent behavior of these tools is to allocate container workloads across a set of available machines in an efficient way, thereby reducing the cost of ownership. Simple to extremely complicated algorithms can be used effectively to improve utilization.
- **Resource allocation:** It allocates servers based on resource availability and the constraints set by application developers. Resource allocation is based on these constraints, affinity rules, port requirements, application dependencies, health, and so on.
- **Service availability:** It ensures that the services are up and running somewhere in the cluster. In case of a machine failure, cluster control tools automatically handle failures by restarting these services on some other machine in the cluster.
- **Agility:** These tools are capable of quickly allocating workloads to the available resources or moving the workload across machines if there is change in resource requirements. Also, constraints can be set to realign the resources based on business criticality, business priority, and so on.
- **Isolation:** Some of these tools provide resource isolation out of the box. Hence, even if the application is not containerized, resource isolation can be still achieved.

A variety of algorithms are used for resource allocation, ranging from simple algorithms to complex algorithms, with machine learning and artificial intelligence. The common algorithms used are random, bin packing, and spread. Constraints set against applications will override the default algorithms based on resource availability:



The preceding diagram shows how these algorithms fill the available machines with deployments. In this case, it is demonstrated with two machines:

- **Spread:** This algorithm performs the allocation of workload equally across the available machines. This is showed in diagram **A**.
- **Bin packing:** This algorithm tries to fill in data machine by machine and ensures the maximum utilization of machines. Bin packing is especially good when using cloud services in a pay-as-you-use style. This is shown in diagram **B**.
- **Random:** This algorithm randomly chooses machines and deploys containers on randomly selected machines. This is showed in diagram **C**.

There is a possibility of using cognitive computing algorithms such as machine learning and collaborative filtering to improve efficiency. Techniques such as **oversubscription** allow a better utilization of resources by allocating underutilized resources for high-priority tasks—for example, revenue-generating services for best-effort tasks such as analytics, video, image processing, and so on.



## Relationship with microservices

The infrastructure of microservices, if not properly provisioned, can easily result in oversized infrastructures and, essentially, a higher cost of ownership. As discussed in the previous sections, a cloud-like environment with a cluster management tool is essential to realize cost benefits when dealing with large-scale microservices

The Spring Boot microservices turbocharged with the Spring Cloud project is the ideal candidate workload to leverage cluster management tools. As Spring Cloud-based microservices are location unaware, these services can be deployed anywhere in the cluster. Whenever services come up, they automatically register to the service registry and advertise their availability. On the other hand, consumers always look for the registry to discover the available service instances. This way, the application supports a full fluid structure without preassuming a deployment topology. With Docker, we were able to abstract the runtime so that the services could run on any Linux-based environments.

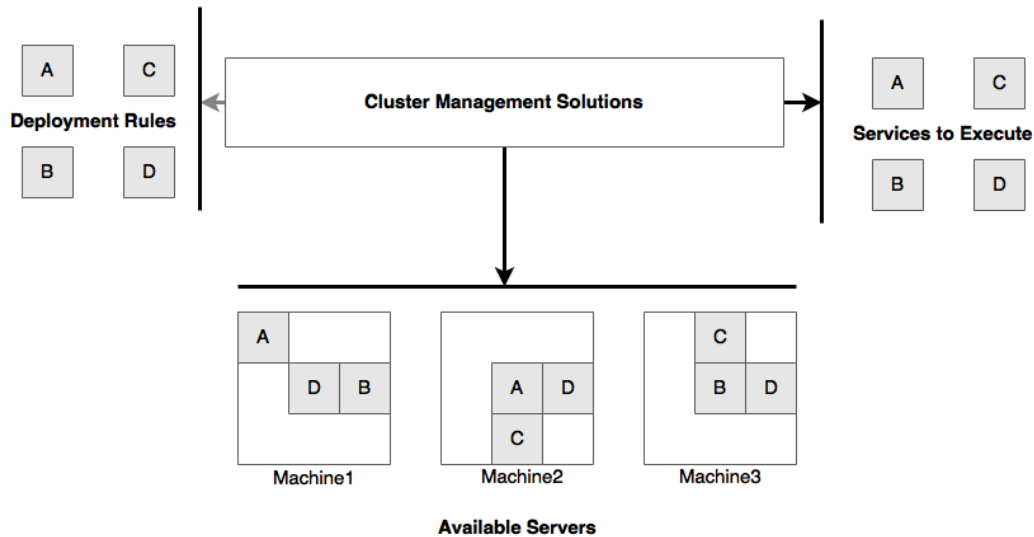
## Relationship with virtualization

Cluster management solutions are different from server virtualization solutions in many aspects. Cluster management solutions run on top of VMs or physical machines as an application component.

## Cluster management solutions

There are many cluster management software tools available. It is unfair to do an apple-to-apple comparison between them. Even though there are no one-to-one components, there are many areas of overlap in capabilities between them. In many situations, organizations use a combination of one or more of these tools to fulfill their requirements.

The following diagram shows the position of cluster management tools from the microservices context:



In this section, we will explore some of the popular cluster management solutions available on the market.

## Docker Swarm

Docker Swarm is Docker's native cluster management solution. Swarm provides a native and deeper integration with Docker and exposes APIs that are compatible with Docker's remote APIs. Docker Swarm logically groups a pool of Docker hosts and manages them as a single large Docker virtual host. Instead of application administrators and developers deciding on which host the container is to be deployed in, this decision making will be delegated to Docker Swarm. Docker Swarm will decide which host to be used based on the bin packing and spread algorithms.

As Docker Swarm is based on Docker's remote APIs, its learning curve for those already using Docker is narrower compared to any other container orchestration tools. However, Docker Swarm is a relatively new product on the market, and it only supports Docker containers.

Docker Swarm works with the concepts of **manager** and **nodes**. A manager is the single point for administrations to interact and schedule the Docker containers for execution. Nodes are where Docker containers are deployed and run.

## Kubernetes

Kubernetes (k8s) comes from Google's engineering, is written in the Go language, and is battle-tested for large-scale deployments at Google. Similar to Swarm, Kubernetes helps manage containerized applications across a cluster of nodes. Kubernetes helps automate container deployments, scheduling, and the scalability of containers. Kubernetes supports a number of useful features out of the box, such as automatic progressive rollouts, versioned deployments, and container resiliency if containers fail due to some reason.

The Kubernetes architecture has the concepts of **master**, **nodes**, and **Pods**. The master and nodes together form a Kubernetes cluster. The master node is responsible for allocating and managing workload across a number of nodes. Nodes are nothing but a VM or a physical machine. Nodes are further subsegmented as pods. A node can host multiple pods. One or more containers are grouped and executed inside a pod. Pods are also helpful in managing and deploying co-located services for efficiency. Kubernetes also supports the concept of labels as key-value pairs to query and find containers. Labels are user-defined parameters to tag certain types of nodes that execute a common type of workloads, such as frontend web servers. The services deployed on a cluster get a single IP/DNS to access the service.

Kubernetes has out-of-the-box support for Docker; however, the Kubernetes learning curve is steeper compared to Docker Swarm. RedHat offers commercial support for Kubernetes as part of its OpenShift platform.

## Apache Mesos

Mesos is an open source framework originally developed by the University of California at Berkeley and is used by Twitter at scale. Twitter uses Mesos primarily to manage the large Hadoop ecosystem.

Mesos is slightly different from the previous solutions. Mesos is more of a resource manager that relays on other frameworks to manage workload execution. Mesos sits between the operating system and the application, providing a logical cluster of machines.

Mesos is a distributed system kernel that logically groups and virtualizes many computers to a single large machine. Mesos is capable of grouping a number of heterogeneous resources to a uniform resource cluster on which applications can be deployed. For these reasons, Mesos is also known as a tool to build a private cloud in a data center.

Mesos has the concepts of the **master** and **slave** nodes. Similar to the earlier solutions, master nodes are responsible for managing the cluster, whereas slaves run the workload. Mesos internally uses ZooKeeper for cluster coordination and storage. Mesos supports the concept of frameworks. These frameworks are responsible for scheduling and running noncontainerized applications and containers. Marathon, Chronos, and Aurora are popular frameworks for the scheduling and execution of applications. Netflix Fenzo is another open source Mesos framework. Interestingly, Kubernetes also can be used as a Mesos framework.

Marathon supports the Docker container as well as noncontainerized applications. Spring Boot can be directly configured in Marathon. Marathon provides a number of capabilities out of the box, such as supporting application dependencies, grouping applications to scale and upgrade services, starting and shutting down healthy and unhealthy instances, rolling out promotes, rolling back failed promotes, and so on.

Mesosphere offers commercial support for Mesos and Marathon as part of its DCOS platform.

## Nomad

Nomad from HashiCorp is another cluster management software. Nomad is a cluster management system that abstracts lower-level machine details and their locations. Nomad has a simpler architecture compared to the other solutions explored earlier. Nomad is also lightweight. Similar to other cluster management solutions, Nomad takes care of resource allocation and the execution of applications. Nomad also accepts user-specific constraints and allocates resources based on this

Nomad has the concept of **servers**, in which all jobs are managed. One server acts as the **leader**, and others act as **followers**. Nomad has the concept of **tasks**, which is the smallest unit of work. Tasks are grouped into **task groups**. A task group has tasks that are to be executed in the same location. One or more task groups or tasks are managed as **jobs**.

Nomad supports many workloads, including Docker, out of the box. Nomad also supports deployments across data centers and is region and data center aware.

## Fleet

Fleet is a cluster management system from CoreOS. It runs on a lower level and works on top of systemd. Fleet can manage application dependencies and make sure that all the required services are running somewhere in the cluster. If a service fails, it restarts the service on another host. Affinity and constraint rules are possible to supply when allocating resources.

Fleet has the concepts of **engine** and **agents**. There is only one engine at any point in the cluster with multiple agents. Tasks are submitted to the engine and agent run these tasks on a cluster machine.

Fleet also supports Docker out of the box.

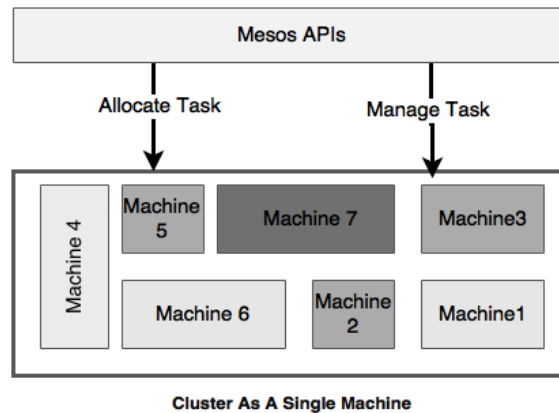
## Cluster management with Mesos and Marathon

As we discussed in the previous section, there are many cluster management solutions or container orchestration tools available. Different organizations choose different solutions to address problems based on their environment. Many organizations choose Kubernetes or Mesos with a framework such as Marathon. In most cases, Docker is used as a default containerization method to package and deploy workloads.

For the rest of this chapter, we will show how Mesos works with Marathon to provide the required cluster management capability. Mesos is used by many organizations, including Twitter, Airbnb, Apple, eBay, Netflix, PayPal, Uber, Yelp, and many others.

## Diving deep into Mesos

Mesos can be treated as a data center kernel. DCOS is the commercial version of Mesos supported by Mesosphere. In order to run multiple tasks on one node, Mesos uses resource isolation concepts. Mesos relies on the Linux kernel's **cgroups** to achieve resource isolation similar to the container approach. It also supports containerized isolation using Docker. Mesos supports both batch workload as well as the OLTP kind of workloads:

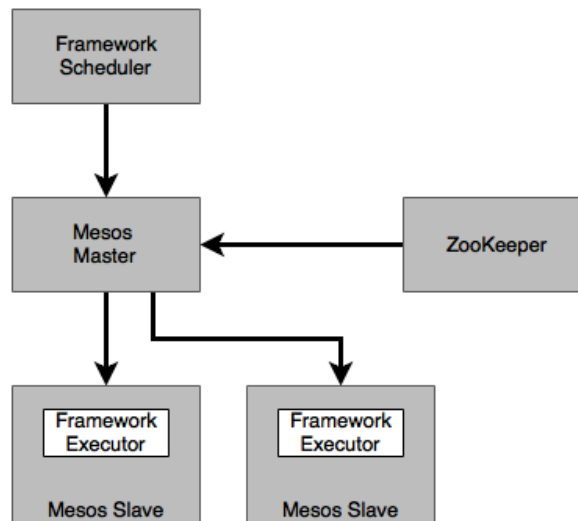


Mesos is an open source top-level Apache project under the Apache license. Mesos abstracts lower-level computing resources such as CPU, memory, and storage from lower-level physical or virtual machines.

Before we examine why we need both Mesos and Marathon, let's understand the Mesos architecture.

## The Mesos architecture

The following diagram shows the simplest architectural representation of Mesos. The key components of Mesos includes a Mesos master node, a set of slave nodes, a ZooKeeper service, and a Mesos framework. The Mesos framework is further subdivided into two components: a scheduler and an executor:



The boxes in the preceding diagram are explained as follows:

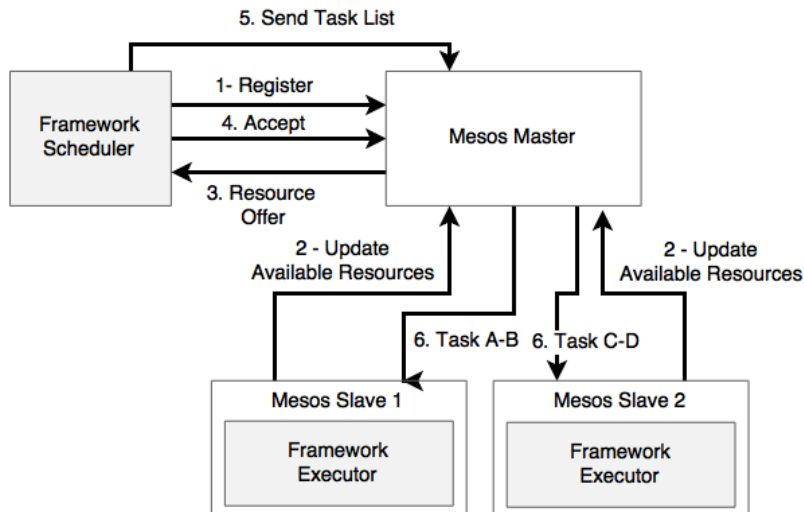
- **Master:** The Mesos master is responsible for managing all the Mesos slaves. The Mesos master gets information on the resource availability from all slave nodes and take the responsibility of filling the resources appropriately based on certain resource policies and constraints. The Mesos master preempts available resources from all slave machines and pools them as a single large machine. The master offers resources to frameworks running on slave machines based on this resource pool.

For high availability, the Mesos master is supported by the Mesos master's standby components. Even if the master is not available, the existing tasks can still be executed. However, new tasks cannot be scheduled in the absence of a master node. The master standby nodes are nodes that wait for the failure of the active master and take over the master's role in the case of a failure. It uses ZooKeeper for the master leader election. A minimum quorum requirement must be met for leader election.

- **Slave:** Mesos slaves are responsible for hosting task execution frameworks. Tasks are executed on the slave nodes. Mesos slaves can be started with attributes as key-value pairs, such as *data center = X*. This is used for constraint evaluations when deploying workloads. Slave machines share resource availability with the Mesos master.
- **ZooKeeper:** ZooKeeper is a centralized coordination server used in Mesos to coordinate activities across the Mesos cluster. Mesos uses ZooKeeper for leader election in case of a Mesos master failure.
- **Framework:** The Mesos framework is responsible for understanding the application's constraints, accepting resource offers from the master, and finally running tasks on the slave resources offered by the master. The Mesos framework consists of two components: the framework scheduler and the framework executor:
  - The scheduler is responsible for registering to Mesos and handling resource offers
  - The executor runs the actual program on Mesos slave nodes

The framework is also responsible for enforcing certain policies and constraints. For example, a constraint can be, let's say, that a minimum of 500 MB of RAM is available for execution.

Frameworks are pluggable components and are replaceable with another framework. The framework workflow is depicted in the following diagram:



The steps denoted in the preceding workflow diagram are elaborated as follows

1. The framework registers with the Mesos master and waits for resource offers. The scheduler may have many tasks in its queue to be executed with different resource constraints (tasks **A** to **D**, in this example). A task, in this case, is a unit of work that is scheduled—for example, a Spring Boot microservice.
2. The Mesos slave offers the available resources to the Mesos master. For example, the slave advertises the CPU and memory available with the slave machine.
3. The Mesos master then creates a resource offer based on the allocation policies set and offers it to the scheduler component of the framework. Allocation policies determine which framework the resources are to be offered to and how many resources are to be offered. The default policies can be customized by plugging additional allocation policies.
4. The scheduler framework component, based on the constraints, capabilities, and policies, may accept or reject the resource offering. For example, a framework rejects the resource offer if the resources are insufficient as per the constraints and policies set.



5. If the scheduler component accepts the resource offer, it submits the details of one more task to the Mesos master with resource constraints per task. Let's say, in this example, that it is ready to submit tasks **A** to **D**.
6. The Mesos master sends this list of tasks to the slave where the resources are available. The framework executor component installed on the slave machines picks up and runs these tasks.

Mesos supports a number of frameworks, such as:

- Marathon and Aurora for **long-running** processes, such as web applications
- Hadoop, Spark, and Storm for **big data** processing
- Chronos and Jenkins for **batch scheduling**
- Cassandra and Elasticsearch for **data management**

In this chapter, we will use Marathon to run dockerized microservices.

## Marathon

Marathon is one of the Mesos framework implementations that can run both container as well as noncontainer execution. Marathon is particularly designed for long-running applications, such as a web server. Marathon ensures that the service started with Marathon continues to be available even if the Mesos slave it is hosted on fails. This will be done by starting another instance.

Marathon is written in Scala and is highly scalable. Marathon offers a UI as well as REST APIs to interact with Marathon, such as the start, stop, scale, and monitoring applications.

Similar to Mesos, Marathon's high availability is achieved by running multiple Marathon instances pointing to a ZooKeeper instance. One of the Marathon instances acts as a leader, and others are in standby mode. In case the leading master fails, a leader election will take place, and the next active master will be determined.

Some of the basic features of Marathon include:

- Setting resource constraints
- Scaling up, scaling down, and the instance management of applications
- Application version management
- Starting and killing applications

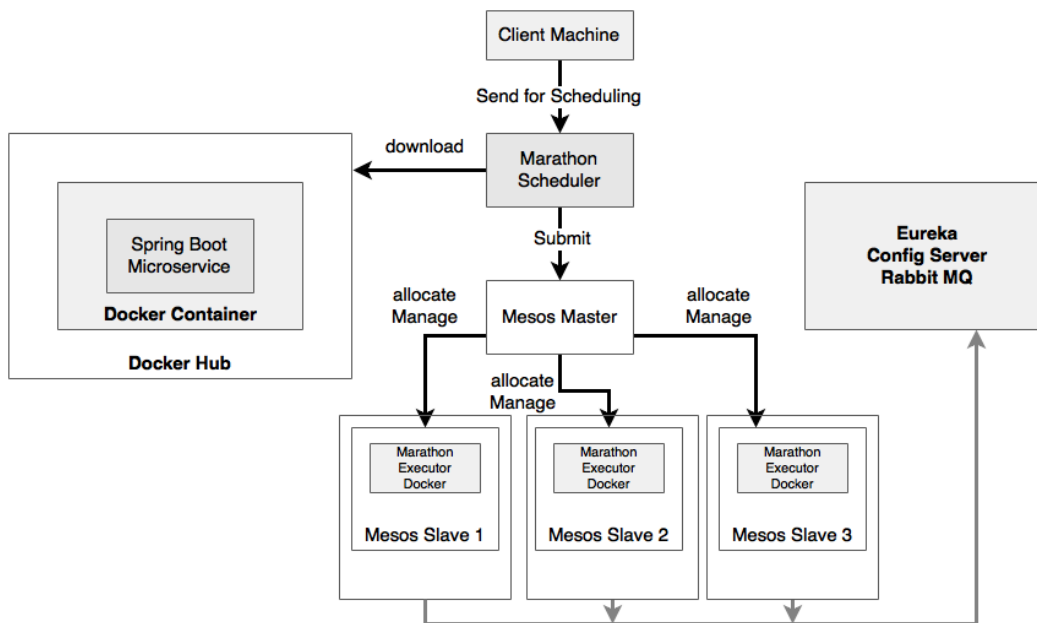
Some of the advanced features of Marathon include:

- Rolling upgrades, rolling restarts, and rollbacks
- Blue-green deployments

## Implementing Mesos and Marathon for BrownField microservices

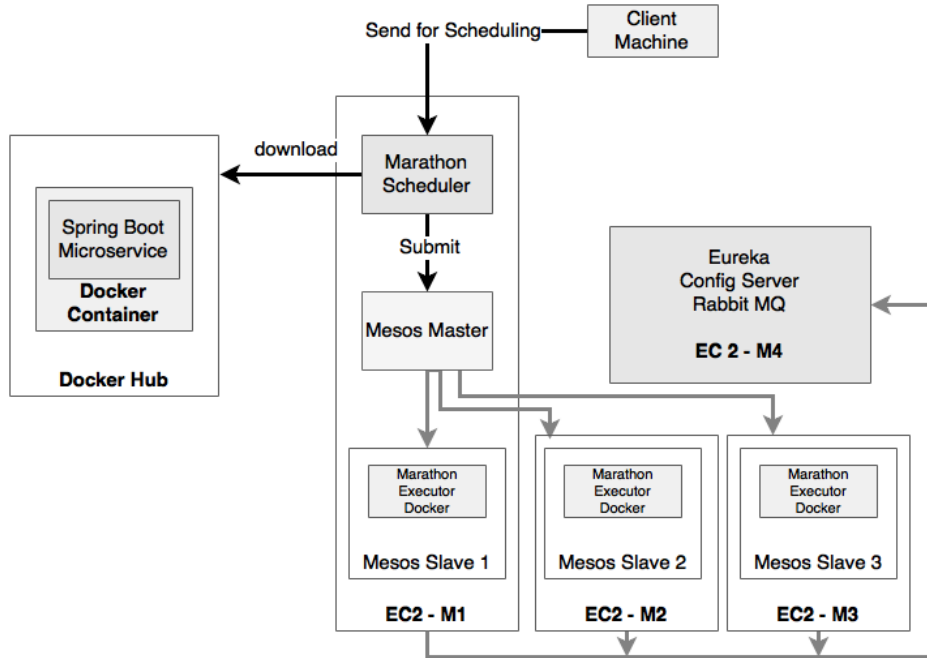
In this section, the dockerized Brownfield microservice developed in *Chapter 8, Containerizing Microservices with Docker*, will be deployed into the AWS cloud and managed with Mesos and Marathon.

For the purposes of demonstration, only three of the services (**Search**, **Search API Gateway**, and **Website**) are covered in the explanations:



The logical architecture of the target state implementation is shown in the preceding diagram. The implementation uses multiple Mesos slaves to execute dockerized microservices with a single Mesos master. The Marathon scheduler component is used to schedule dockerized microservices. Dockerized microservices are hosted on the Docker Hub registry. Dockerized microservices are implemented using Spring Boot and Spring Cloud.

The following diagram shows the physical deployment architecture:



As shown in the preceding diagram, in this example, we will use four EC2 instances:

- **EC2-M1**: This hosts the Mesos master, ZooKeeper, the Marathon scheduler, and one Mesos slave instance
- **EC2-M2**: This hosts one Mesos slave instance
- **EC2-M3**: This hosts another Mesos slave instance
- **EC2-M4**: This hosts Eureka, Config server, and RabbitMQ

For a real production setup, multiple Mesos masters as well as multiple instances of Marathon are required for fault tolerance.

## Setting up AWS

Launch the four **t2.micro** EC2 instances that will be used for this deployment. All four instances have to be on the same security group so that the instances can see each other using their local IP addresses.

The following tables show the machine details and IP addresses for indicative purposes and to link subsequent instructions:

Launch Instance

Connect

Actions ▾

Filter by tags and attributes or search by keyword

<input type="checkbox"/>	Name ▾	Instance ID ▾	Instance Type ▾	Availability Zone ▾	Instance State ▾
<input type="checkbox"/>	mesos-master slave1	i-06100786	t2.micro	us-east-1b	<span style="color: green;">●</span> running
<input type="checkbox"/>	config-eureka-rabbit	i-2404e5a7	t2.micro	us-east-1b	<span style="color: green;">●</span> running
<input type="checkbox"/>	mesos-slave2	i-a7df2b3a	t2.micro	us-east-1b	<span style="color: green;">●</span> running
<input checked="" type="checkbox"/>	mesos-slave3	i-b0eb1f2d	t2.micro	us-east-1b	<span style="color: green;">●</span> running

Instance ID	Private DNS/IP	Public DNS/IP
i-06100786	ip-172-31-54-69.ec2. internal 172.31.54.69	ec2-54-85-107-37.compute-1. amazonaws.com 54.85.107.37
i-2404e5a7	ip-172-31-62-44.ec2. internal 172.31.62.44	ec2-52-205-251-150.compute-1. amazonaws.com 52.205.251.150
i-a7df2b3a	ip-172-31-49-55.ec2. internal 172.31.49.55	ec2-54-172-213-51.compute-1. amazonaws.com 54.172.213.51
i-b0eb1f2d	ip-172-31-53-109.ec2. internal 172.31.53.109	ec2-54-86-31-240.compute-1. amazonaws.com 54.86.31.240

Replace the IP and DNS addresses based on your AWS EC2 configuration

## Installing ZooKeeper, Mesos, and Marathon

The following software versions will be used for the deployment. The deployment in this section follows the physical deployment architecture explained in the earlier section:

- Mesos version 0.27.1
- Docker version 1.6.2, build 7c8fca2
- Marathon version 0.15.3



The detailed instructions to set up ZooKeeper, Mesos, and Marathon are available at <https://open.mesosphere.com/getting-started/install/>.

Perform the following steps for a minimal installation of ZooKeeper, Mesos, and Marathon to deploy the BrownField microservice:

1. As a prerequisite, JRE 8 must be installed on all the machines. Execute the following command:

```
sudo apt-get -y install oracle-java8-installer
```

2. Install Docker on all machines earmarked for the Mesos slave via the following command:

```
sudo apt-get install docker
```

3. Open a terminal window and execute the following commands. These commands set up the repository for installation:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
E56151BF
DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
CODENAME=$(lsb_release -cs)
Add the repository
echo "deb http://repos.mesosphere.com/${DISTRO} ${CODENAME} main"
| \
 sudo tee /etc/apt/sources.list.d/mesosphere.list
sudo apt-get -y update
```

4. Execute the following command to install Mesos and Marathon. This will also install Zookeeper as a dependency:

```
sudo apt-get -y install mesos marathon
```

Repeat the preceding steps on all the three EC2 instances reserved for the Mesos slave execution. As the next step, ZooKeeper and Mesos have to be configured on the machine identified for the Mesos master

## Configuring ZooKeeper

Connect to the machine reserved for the Mesos master and Marathon scheduler. In this case, 172.31.54.69 will be used to set up ZooKeeper, the Mesos master, and Marathon.

There are two configuration changes required in ZooKeeper, as follows

1. The first step is to set `/etc/zookeeper/conf/myid` to a unique integer between 1 and 255, as follows:
2. The next step is to edit `/etc/zookeeper/conf/zoo.cfg`. Update the file to reflect the following changes

```
Open vi /etc/zookeeper/conf/myid and set 1.
```

```
specify all zookeeper servers
The first port is used by followers to connect to the leader
The second one is used for leader election
server.1= 172.31.54.69:2888:3888
#server.2=zookeeper2:2888:3888
#server.3=zookeeper3:2888:3888
```

Replace the IP addresses with the relevant private IP address. In this case, we will use only one ZooKeeper server, but in a production scenario, multiple servers are required for high availability.

## Configuring Mesos

Make changes to the Mesos configuration to point to ZooKeeper, set up a quorum, and enable Docker support via the following steps:

1. Edit `/etc/mesos/zk` to set the following value. This is to point Mesos to a ZooKeeper instance for quorum and leader election:
2. Edit the `/etc/mesos-master/quorum` file and set the value as 1. In a production scenario, we may need a minimum quorum of three:

```
zk:// 172.31.54.69:2181/mesos
```

```
vi /etc/mesos-master/quorum
```

3. The default Mesos installation does not support Docker on Mesos slaves. In order to enable Docker, update the following `mesos-slave` configuration

```
echo 'docker,mesos' > /etc/mesos-slave/containerizers
```

## Running Mesos, Marathon, and ZooKeeper as services

All the required configuration changes are implemented. The easiest way to start Mesos, Marathon, and Zookeeper is to run them as services, as follows:

- The following commands start services. The services need to be started in the following order:

```
sudo service zookeeper start
sudo service mesos-master start
sudo service mesos-slave start
sudo service marathon start
```

- At any point, the following commands can be used to stop these services:

```
sudo service zookeeper stop
sudo service mesos-master stop
sudo service mesos-slave stop
sudo service marathon stop
```

- Once the services are up and running, use a terminal window to verify whether the services are running:

```
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep zookeeper
ubuntu 1240 1180 0 16:33 pts/0 00:00:00 grep --color=auto zookeeper
zookeeper+ 17056 1 0 Apr10 ? 00:00:54 /usr/bin/java -cp /etc/zookeeper/conf:/usr/share/java/jline.jar:/usr/share/java/log4j-1.2.jar:/usr/share/java/xercesImpl.jar:/usr/share/java/xmlParserAPIs.jar:/usr/share/java/netty.jar:/usr/share/java/slf4j-api.jar:/usr/share/java/slf4j-log4j12.jar:/usr/share/java/zookeeper.jar -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.local.only=false -Dzookeeper.log.dir=/var/log/zookeeper -Dzookeeper.root.logger=INFO,ROLLINGFILE org.apache.zookeeper.server.quorum.QuorumPeerMain /etc/zookeeper/conf/zoo.cfg
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep sbin/mesos-master
ubuntu 1245 1180 0 16:33 pts/0 00:00:00 grep --color=auto sbin/mesos-master
root 17066 1 0 Apr10 ? 00:00:28 /usr/sbin/mesos-master --zk=zk://172.31.54.69:2181/mesos --port=5050 --log_dir=/var/log/mesos --quorum=1 --work_dir=/var/lib/mesos
ubuntu@ip-172-31-54-69:~$
ubuntu@ip-172-31-54-69:~$ ps -ef | grep marathon.Main
ubuntu 1252 1180 0 16:33 pts/0 00:00:00 grep --color=auto marathon.Main
root 17117 1 0 Apr10 ? 00:04:07 java -Djava.library.path=/usr/local/lib:/usr/lib:/usr/lib64 -Djava.util.logging.SimpleFormatter.format=%2$s%5$s%6$s%n -Xmx512m -cp /usr/bin/marathon mesosphere.marathon.Main --zk zk://172.31.54.69:2181/marathon --master zk://172.31.54.69:2181/mesos
ubuntu@ip-172-31-54-69:~$ █
```

## Running the Mesos slave in the command line

In this example, instead of using the Mesos slave service, we will use a command-line version to invoke the Mesos slave to showcase additional input parameters. Stop the Mesos slave and use the command line as mentioned here to start the slave again:

```
$sudo service mesos-slave stop
```

```
$sudo /usr/sbin/mesos-slave --master=172.31.54.69:5050 --log_dir=/var/
log/mesos --work_dir=/var/lib/mesos --containerizers=mesos,docker --resou
rces="ports(*):[8000-9000, 31000-32000]"
```

The command-line parameters used are explained as follows:

- `--master=172.31.54.69:5050`: This parameter is to tell the Mesos slave to connect to the correct Mesos master. In this case, there is only one master running at 172.31.54.69:5050. All the slaves connect to the same Mesos master.
- `--containerizers=mesos,docker`: This parameter is to enable support for Docker container execution as well as noncontainerized executions on the Mesos slave instances.
- `--resources="ports(*):[8000-9000, 31000-32000]"`: This parameter indicates that the slave can offer both ranges of ports when binding resources. 31000 to 32000 is the default range. As we are using port numbers starting with 8000, it is important to tell the Mesos slave to allow exposing ports starting from 8000 as well.

Perform the following steps to verify the installation of Mesos and Marathon:

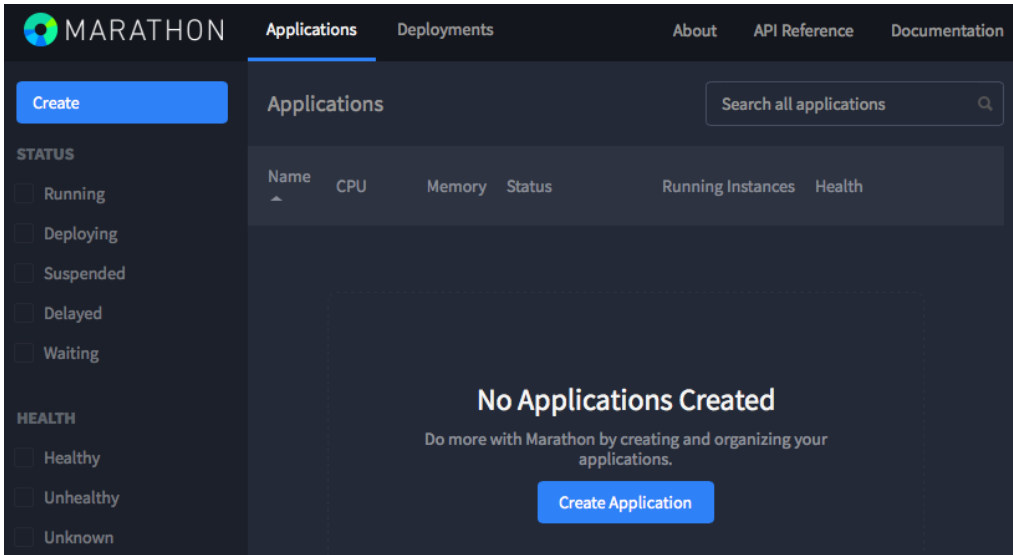
1. Execute the command mentioned in the previous step to start the Mesos slave on all the three instances designated for the slave. The same command can be used across all three instances as all of them connect to the same master.
2. If the Mesos slave is successfully started, a message similar to the following will appear in the console:

```
I0411 18:11:39.684809 16665 slave.cpp:1030] Forwarding total
oversubscribed resources
```

The preceding message indicates that the Mesos slave started sending the current state of resource availability periodically to the Mesos master.

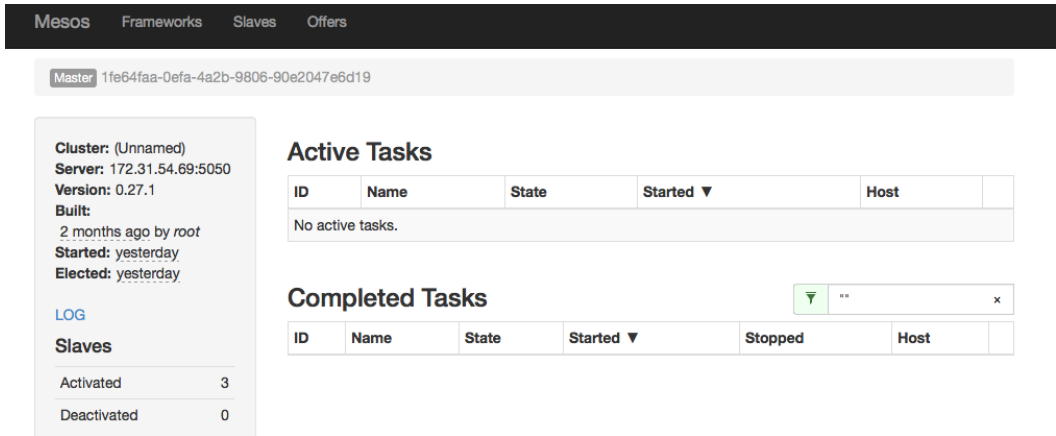


3. Open `http://54.85.107.37:8080` to inspect the Marathon UI. Replace the IP address with the public IP address of the EC2 instance:



As there are no applications deployed so far, the **Applications** section of the UI is empty.


4. Open the Mesos UI, which runs on port 5050, by going to `http://54.85.107.37:5050`:



The **Slaves** section of the console shows that there are three activated Mesos slaves available for execution. It also indicates that there is no active task.

## Preparing BrownField PSS services

In the previous section, we successfully set up Mesos and Marathon. In this section, we will take a look at how to deploy the BrownField PSS application previously developed using Mesos and Marathon.

 The full source code of this chapter is available under the Chapter 9 project in the code files. Copy `chapter8.configserver`, `chapter8.eurekaclient`, `chapter8.search`, `chapter8.search-apigateway`, and `chapter8.website` into a new STS workspace and rename them `chapter9.*`.

1. Before we deploy any application, we have to set up the Config server, Eureka server, and RabbitMQ in one of the servers. Follow the steps described in the *Running BrownField services on EC2* section in *Chapter 8, Containerizing Microservices with Docker*. Alternately, we can use the same instance as used in the previous chapter for this purpose.
2. Change all `bootstrap.properties` files to reflect the Config serv IP address.
3. Before we deploy our services, there are a few specific changes required on the microservices. When running dockerized microservices with the BRIDGE mode on, we need to tell the Eureka client the hostname to be used to bind. By default, Eureka uses the **instance ID** to register. However, this is not helpful as Eureka clients won't be able to look up these services using the instance ID. In the previous chapter, the HOST mode was used instead of the BRIDGE mode.

The hostname setup can be done using the `eureka.instance.hostname` property. However, when running on AWS specifically, an alternate approach is to define a bean in the microservices to pick up AWS-specific information, as follows:

```
@Configuration
class EurekaConfig {
 @Bean
 public EurekaInstanceConfigBean eurekaInstanceConfigBean() {
 EurekaInstanceConfigBean config = new
 EurekaInstanceConfigBean(new InetUtils(new
 InetUtilsProperties()));
 AmazonInfo info = AmazonInfo.Builder.newBuilder().
 autoBuild("eureka");
 config.setDataCenterInfo(info);
 }
}
```

```
 info.getMetadata().put(AmazonInfo.MetadataKey.
publicHostname.getName(), info.get(AmazonInfo.MetadataKey.
publicIpv4));
 config.setHostname(info.get(AmazonInfo.MetadataKey.
localHostname));
config.setNonSecurePortEnabled(true);
config.setNonSecurePort(PORT);
config.getMetadataMap().put("instanceId", info.get(AmazonInfo.
MetadataKey.localHostname));
return config;
}
```

The preceding code provides a custom Eureka server configuration using the Amazon host information using Netflix APIs. The code overrides the hostname and instance ID with the private DNS. The port is read from the Config server. This code also assumes one host per service so that the port number stays constant across multiple deployments. This can also be overridden by dynamically reading the port binding information at runtime.

The previous code has to be applied in all microservices.

4. Rebuild all the microservices using Maven. Build and push the Docker images to the Docker Hub. The steps for the three services are shown as follows. Repeat the same steps for all the other services. The working directory needs to be switched to the respective directories before executing these commands:

```
docker build -t search-service:1.0 .
docker tag search-service:1.0 rajeshrv/search-service:1.0
docker push rajeshrv/search-service:1.0

docker build -t search-apigateway:1.0 .
docker tag search-apigateway:1.0 rajeshrv/search-apigateway:1.0
docker push rajeshrv/search-apigateway:1.0

docker build -t website:1.0 .
docker tag website:1.0 rajeshrv/website:1.0
docker push rajeshrv/website:1.0
```

## Deploying BrownField PSS services

The Docker images are now published to the Docker Hub registry. Perform the following steps to deploy and run BrownField PSS services:

1. Start the Config server, Eureka server, and RabbitMQ on its dedicated instance.
2. Make sure that the Mesos server and Marathon are running on the machine where the Mesos master is configured
3. Run the Mesos slave on all the machines as described earlier using the command line.
4. At this point, the Mesos Marathon cluster is up and running and is ready to accept deployments. The deployment can be done by creating one JSON file per service, as shown here:

```
{
 "id": "search-service-1.0",
 "cpus": 0.5,
 "mem": 256.0,
 "instances": 1,
 "container": {
 "docker": {
 "type": "DOCKER",
 "image": "rajeshrv/search-service:1.0",
 "network": "BRIDGE",
 "portMappings": [
 { "containerPort": 0, "hostPort": 8090 }
]
 }
 }
}
```

The preceding JSON code will be stored in the `search.json` file. Similarly, create a JSON file for other services as well.

The JSON structure is explained as follows:

- `id`: This is the unique ID of the application. This can be a logical name.
- `cpus` and `mem`: This sets the resource constraints for this application. If the resource offer does not satisfy this resource constraint, Marathon will reject this resource offer from the Mesos master.
- `instances`: This decides how many instances of this application to start with. In the preceding configuration, by default, it starts one instance as soon as it gets deployed. Marathon maintains the number of instances mentioned at any point.

- **container:** This parameter tells the Marathon executor to use a Docker container for execution.
  - **image:** This tells the Marathon scheduler which Docker image has to be used for deployment. In this case, this will download the `search-service:1.0` image from the Docker Hub repository `rajeshrv`.
  - **network:** This value is used for Docker runtime to advise on the network mode to be used when starting the new docker container. This can be `BRIDGE` or `HOST`. In this case, the `BRIDGE` mode will be used.
  - **portMappings:** The port mapping provides information on how to map the internal and external ports. In the preceding configuration, the host port is set as `8090`, which tells the Marathon executor to use `8090` when starting the service. As the container port is set as `0`, the same host port will be assigned to the container. Marathon picks up random ports if the host port value is `0`.
5. Additional health checks are also possible with the JSON descriptor, as shown here:

```
"healthChecks": [
 {
 "protocol": "HTTP",
 "portIndex": 0,
 "path": "/admin/health",
 "gracePeriodSeconds": 100,
 "intervalSeconds": 30,
 "maxConsecutiveFailures": 5
 }
]
```

6. Once this JSON code is created and saved, deploy it to Marathon using the Marathon REST APIs as follows:

```
curl -X POST http://54.85.107.37:8080/v2/apps -d @search.json -H
"Content-type: application/json"
```

Repeat this step for all the other services as well.

The preceding step will automatically deploy the Docker container to the Mesos cluster and start one instance of the service.

# Reviewing the deployment

The steps for this are as follows:

1. Open the Marathon UI. As shown in the following screenshot, the UI shows that all the three applications are deployed and are in the **Running** state. It also indicates that **1 of 1** instance is in the **Running** state:

The screenshot shows the Marathon UI interface. On the left, there's a sidebar with 'Create' and 'STATUS' sections. The 'STATUS' section shows 'Running' with a count of 3. The main area is titled 'Applications' and contains a table with columns: Name, CPU, Memory, Status, Running Instances, and Health. Three applications are listed: 'search-apigateway-1.0', 'search-service-1.0', and 'website-1.0'. All three are in the 'Running' state with '1 of 1' instances.

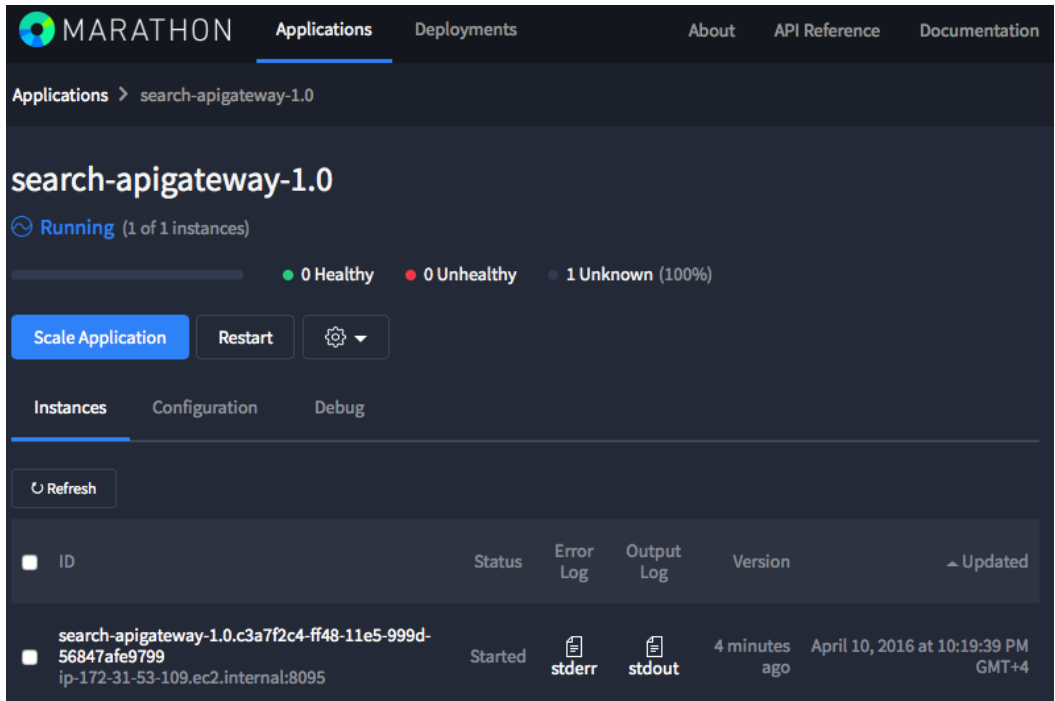
Name	CPU	Memory	Status	Running Instances	Health
search-apigateway-1.0	0.5	256 MiB	Running	1 of 1	...
search-service-1.0	0.5	256 MiB	Running	1 of 1	...
website-1.0	0.5	256 MiB	Running	1 of 1	...

2. Visit the Mesos UI. As shown in the following screenshot, there are three **Active Tasks**, all of them in the **Running** state. It also shows the host in which these services run:

The screenshot shows the Mesos UI interface. On the left, there's a sidebar with 'Cluster: (Unnamed)', 'Server: 172.31.54.69:5050', 'Version: 0.27.1', 'Built: 2 months ago by root', 'Started: 3 hours ago', and 'Elected: 3 hours ago'. Below this is a 'LOG' section and a 'Slaves' section showing 'Activated' (3) and 'Deactivated' (0). The main area is titled 'Active Tasks' and contains a table with columns: ID, Name, State, Started, Host, and a link to 'Sandbox'. Three tasks are listed: 'website-1.0', 'search-apigateway-1.0', and 'search-service-1.0'. All three are in the 'RUNNING' state.

ID	Name	State	Started	Host	Sandbox
website-1.0.fb262fa5-ff48-11e5-999d-56847afe9799	website-1.0	RUNNING	5 minutes ago	ip-172-31-49-55.ec2.internal	Sandbox
search-apigateway-1.0.c3a7f2c4-ff48-11e5-999d-56847afe9799	search-apigateway-1.0	RUNNING	7 minutes ago	ip-172-31-53-109.ec2.internal	Sandbox
search-service-1.0.9e971e23-ff48-11e5-999d-56847afe9799	search-service-1.0	RUNNING	8 minutes ago	ip-172-31-54-69.ec2.internal	Sandbox

3. In the Marathon UI, click on a running application. The following screenshot shows the **search-apigateway-1.0** application. In the **Instances** tab, the IP address and port in which the service is bound is indicated:



The **Scale Application** button allows administrators to specify how many instances of the service are required. This can be used to scale up as well as scale down instances.

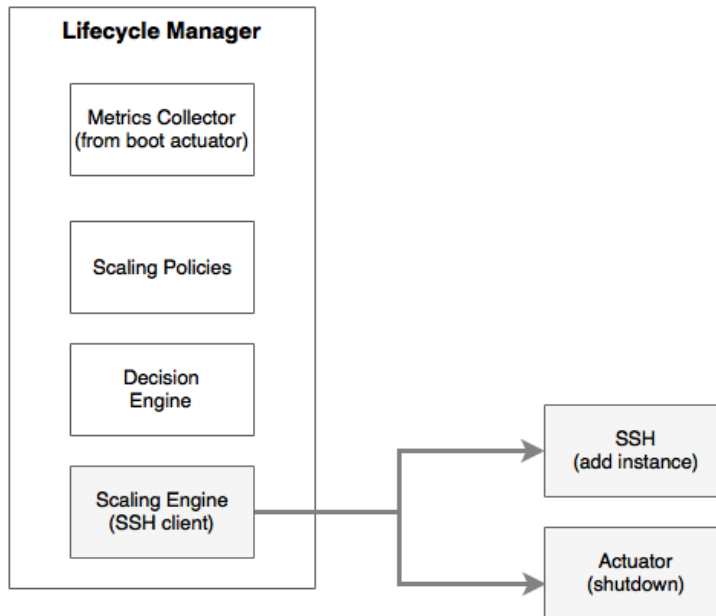
4. Open the Eureka server console to take a look at how the services are bound. As shown in the screenshot, **AMIs** and **Availability Zones** are reflected when services are registered. Follow `http://52.205.251.150:8761`:

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
SEARCH-APIGATEWAY	ami-fce3c696 (1)	us-east-1b (1)	UP (1) - ip-172-31-53-109.ec2.internal
SEARCH-SERVICE	ami-fce3c696 (1)	us-east-1b (1)	UP (1) - ip-172-31-54-69.ec2.internal
TEST-CLIENT	ami-fce3c696 (1)	us-east-1b (1)	UP (1) - ip-172-31-49-55.ec2.internal

5. Open `http://54.172.213.51:8001` in a browser to verify the **Website** application.

## A place for the life cycle manager

The life cycle manager introduced in *Chapter 6, Autoscaling Microservices*, has the capability of autoscaling up or down instances based on demand. It also has the ability to take decisions on where to deploy and how to deploy applications on a cluster of machines based on policies and constraints. The life cycle manager's capabilities are shown in the following figure



Marathon has the capability to manage clusters and deployments to clusters based on policies and constraints. The number of instances can be altered using the Marathon UI.

There are redundant capabilities between our life cycle manager and Marathon. With Marathon in place, SSH work or machine-level scripting is no longer required. Moreover, deployment policies and constraints can be delegated to Marathon. The REST APIs exposed by Marathon can be used to initiate scaling functions.

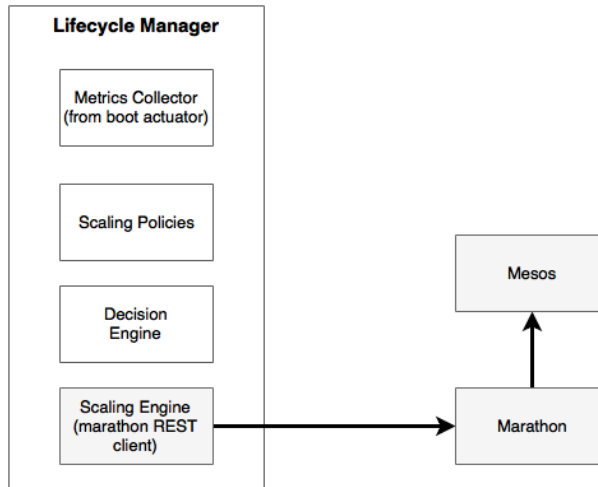
**Marathon autoscale** is a proof-of-concept project from Mesosphere for autoscaling. The Marathon autoscale provides basic autoscale features such as the CPU, memory, and rate of request.



## Rewriting the life cycle manager with Mesos and Marathon

We still need a custom life cycle manager to collect metrics from the Spring Boot actuator endpoints. A custom life cycle manager is also handy if the scaling rules are beyond the CPU, memory, and rate of scaling.

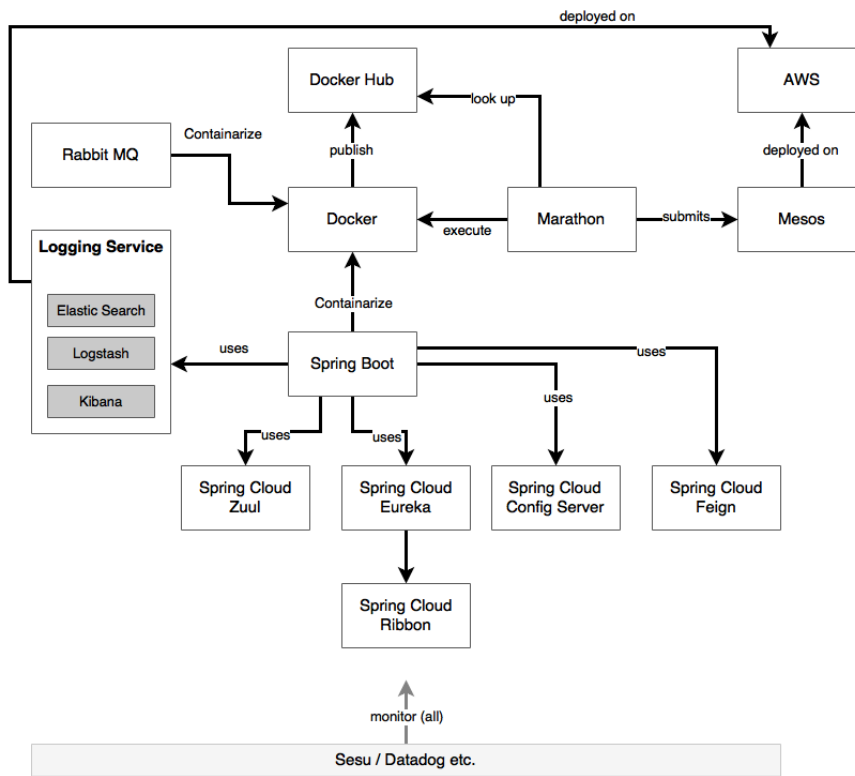
The following diagram shows the updated life cycle manager using the Marathon framework:



The life cycle manager, in this case, collects actuator metrics from different Spring Boot applications, combines them with other metrics, and checks for certain thresholds. Based on the scaling policies, the decision engine informs the scaling engine to either scale down or scale up. In this case, the scaling engine is nothing but a Marathon REST client. This approach is cleaner and neater than our earlier primitive life cycle manager implementation using SSH and Unix scripts.

## The technology metamodel

We have covered a lot of ground on microservices with the BrownField PSS microservices. The following diagram sums it up by bringing together all the technologies used into a technology metamodel:



## Summary

In this chapter, you learned the importance of a cluster management and init system to efficiently manage dockerized microservices at scale

We explored the different cluster control or cluster orchestration tools before diving deep into Mesos and Marathon. We also implemented Mesos and Marathon in the AWS cloud environment to demonstrate how to manage dockerized microservices developed for BrownField PSS.

At the end of this chapter, we also explored the position of the life cycle manager in conjunction with Mesos and Marathon. Finally, we concluded this chapter with a technology metamodel based on the BrownField PSS microservices implementation.

So far, we have discussed all the core and supporting technology capabilities required for a successful microservices implementation. A successful microservice implementation also requires processes and practices beyond technology. The next chapter, the last in the book, will cover the process and practice perspectives of microservices.



# 10

## The Microservices Development Life Cycle

Similar to the **software development life cycle (SDLC)**, it is important to understand the aspects of the microservice development life cycle processes for a successful implementation of the microservices architecture.

This final chapter will focus on the development process and practice of microservices with the help of BrownField Airline's PSS microservices example. Furthermore, this chapter will describe best practices in structuring development teams, development methodologies, automated testing, and continuous delivery of microservices in line with DevOps practices. Finally, this chapter will conclude by shedding light on the importance of the reference architecture in a decentralized governance approach to microservices.

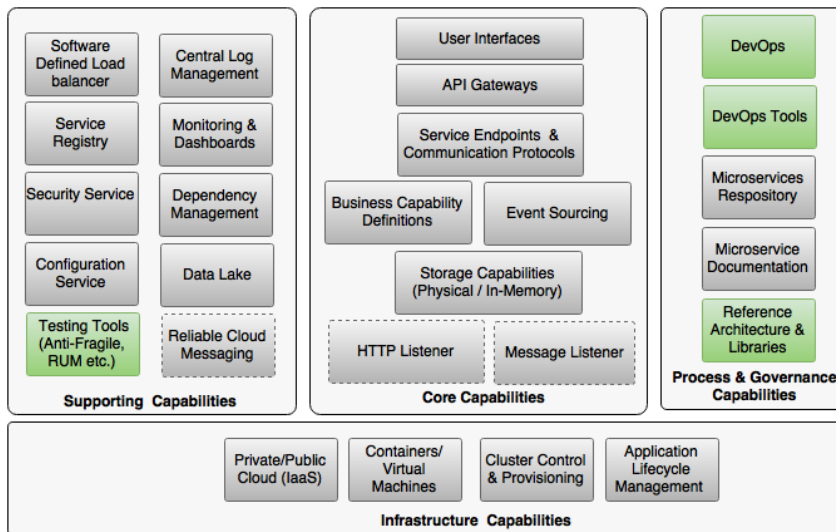
By the end of this chapter, you will learn about the following topics:

- Reviewing DevOps in the context of microservices development
- Defining the microservices life cycle and related processes
- Best practices around the development, testing, and deployment of Internet-scale microservices

# Reviewing the microservice capability model

This chapter will cover the following microservices capabilities from the microservices capability model discussed in *Chapter 3, Applying Microservices Concepts*:

- DevOps
- DevOps Tools
- Reference Architecture & Libraries
- Testing Tools (Anti-Fragile, RUM etc)



## The new mantra of lean IT – DevOps

We discussed the definition of DevOps in *Chapter 2, Building Microservices with Spring Boot*. Here is a quick recap of the DevOps definition

Gartner defines DevOps as follows

*"DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach. DevOps emphasizes people (and culture), and seeks to improve collaboration between operations and development teams. DevOps implementations utilize technology – especially automation tools that can leverage an increasingly programmable and dynamic infrastructure from a life cycle perspective."*

DevOps and microservices evolved independently. *Chapter 1, Demystifying Microservices*, explored the evolution of microservices. In this section, we will review the evolution of DevOps and then take a look at how DevOps supports microservices adoption.

In the era of digital disruption and in order to support modern business, IT organizations have to master two key areas: speed of delivery and value-driven delivery. This is obviously apart from being expert in leading technologies.

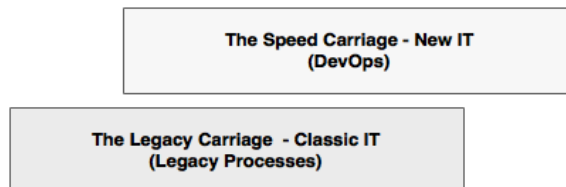
Many IT organizations failed to master this change, causing frustration to business users. To overcome this situation, many business departments started their own shadow IT or stealth IT under their control. Some smart IT organizations then adopted a lean IT model to respond to these situations.

However, many organizations still struggle with this transformation due to the large baggage of legacy systems and processes. Gartner coined the concept of a **pace-layered application strategy**. Gartner's view is that high speed is required only for certain types of applications or certain business areas. Gartner termed this a **system of innovation**. A system of innovation requires rapid innovations compared to a **system of records**. As a system of innovations needs rapid innovation, a lean IT delivery model is essential for such applications. Practitioners evangelized the lean IT model as DevOps.

There are two key strategies used by organizations to adopt DevOps.

Some organizations positioned DevOps as a process to fill the gaps in their existing processes. Such organizations adopted an incremental strategy for their DevOps journey. The adoption path starts with Agile development, then incrementally adopts continuous integration, automated testing, and release to production and then all DevOps practices. The challenge in such organizations is the time to realize the full benefits as well as the mixed culture of people due to legacy processes

Many organizations, therefore, take a disruptive approach to adopt DevOps. This will be achieved by partitioning IT into two layers or even as two different IT units. The high-speed layer of IT uses DevOps-style practices to dramatically change the culture of the organization with no connection to the legacy processes and practices. A selective application cluster will be identified and moved to the new IT based on the business value:



The intention of DevOps is not just to reduce cost. It also enables the business to disrupt competitors by quickly moving ideas to production. DevOps attacks traditional IT issues in multiple ways, as explained here.

## **Reducing wastage**

DevOps processes and practices essentially speed up deliveries which improves quality. The speed of delivery is achieved by cutting IT wastage. This is achieved by avoiding work that adds no value to the business nor to desired business outcomes. IT wastage includes software defects, productivity issues, process overheads, time lag in decision making, time spent in reporting layers, internal governance, overestimation, and so on. By reducing these wastages, organizations can radically improve the speed of delivery. The wastage is reduced by primarily adopting Agile processes, tools, and techniques.

## **Automating every possible step**

By automating the manually executed tasks, one can dramatically improve the speed of delivery as well as the quality of deliverables. The scope of automation goes from planning to customer feedback. Automation reduces the time to move business ideas to production. This also reduces a number of manual gate checks, bureaucratic decision making, and so on. Automated monitoring mechanisms and feedback go back to the development factory, which gets it fixed and quickly moved to production.

## **Value-driven delivery**

DevOps reduces the gap between IT and business through value-driven delivery. Value-driven delivery closely aligns IT to business by understanding true business values and helps the business by quickly delivering these values, which can give a competitive advantage. This is similar to the shadow IT concept, in which IT is collocated with the business and delivers business needs quickly, rather than waiting for heavy project investment-delivery cycles.

Traditionally, IT is partially disconnected from the business and works with IT KPIs, such as the number of successful project deliveries, whereas in the new model, IT shares business KPIs. As an example, a new IT KPI could be that IT helped business to achieve a 10% increase in sales orders or led to 20% increase in customer acquisition. This will shift IT's organizational position from merely a support organization to a business partner.

## Bridging development and operations

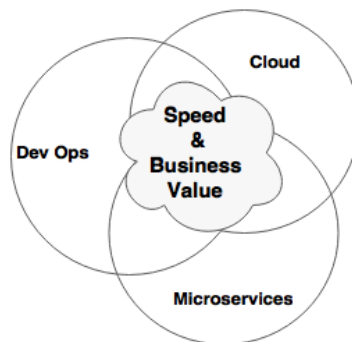
Traditionally, IT has different teams for development and operations. In many cases, they are differentiated with logical barriers. DevOps reduces the gap between the development and operations teams so that it can potentially reduce wastage and improve quality. Multidisciplinary teams work together to address problems at hand rather than throwing mud across the wall.

With DevOps, operations teams will have a fairly good understanding about the services and applications developed by development teams. Similarly, development teams will have a good handle on the infrastructure components and configurations used by the applications. As a result, operations teams can make decisions based exactly on service behaviors rather than enforcing standard organizational policies and rules when designing infrastructure components. This would eventually help the IT organization to improve the quality of the product as well as the time to resolve incidents and problem management.

In the DevOps world, speed of delivery is achieved through the automation of high-velocity changes, and quality is achieved through automation and people. Business values are achieved through efficiency, speed of delivery, quality, and the ability to innovate. Cost reduction is achieved through automation, productivity, and reducing wastage.

## Meeting the trio – microservices, DevOps, and cloud

The trio – cloud, microservices, and DevOps – targets a set of common objectives: speed of delivery, business value, and cost benefit. All three can stay and evolve independently, but they complement each other to achieve the desired common goals. Organizations embarking on any of these naturally tend to consider the other two as they are closely linked together:





Many organizations start their journey with DevOps as an organizational practice to achieve high-velocity release cycles but eventually move to the microservices architecture and cloud. It is not mandatory to have microservices and cloud support DevOps. However, automating the release cycles of large monolithic applications does not make much sense, and in many cases, it would be impossible to achieve. In such scenarios, the microservices architecture and cloud will be handy when implementing DevOps.

If we flip a coin, cloud does not need a microservices architecture to achieve its benefits. However, to effectively implement microservices, both cloud and DevOps are essential.

In summary, if the objective of an organization is to achieve a high speed of delivery and quality in a cost-effective way, the trio together can bring tremendous success.

## Cloud as the self-service infrastructure for Microservices

The main driver for cloud is to improve agility and reduce cost. By reducing the time to provision the infrastructure, the speed of delivery can be increased. By optimally utilizing the infrastructure, one can bring down the cost. Therefore, cloud directly helps achieve both speed of delivery and cost.

As discussed in *Chapter 9, Managing Dockerized Microservices with Mesos and Marathon*, without having a cloud infrastructure with cluster management software, it would be hard to control the infrastructure cost when deploying microservices. Hence, the cloud with self-service capabilities is essential for microservices to achieve their full potential benefits. In the microservices context, the cloud not only helps abstract the physical infrastructure but also provides software APIs for dynamic provisioning and automatic deployments. This is referred to as **infrastructure as code** or **software-defined infrastructure**.

## DevOps as the practice and process for microservices

Microservice is an architecture style that enables quick delivery. However, microservices cannot provide the desired benefits by themselves. A microservices based project with a delivery cycle of 6 months does not give the targeted speed of delivery or business agility. Microservices need a set of supporting delivery practices and processes to effectively achieve their goal.

DevOps is the ideal candidate for the underpinning process and practices for microservice delivery. DevOps processes and practices gel well with the microservices architecture's philosophies.

## Practice points for microservices development

For a successful microservice delivery, a number of development-to-delivery practices need to be considered, including the DevOps philosophy. In the previous chapters, you learned the different architecture capabilities of microservices. In this section, we will explore the nonarchitectural aspects of microservice developments.

## Understanding business motivation and value

Microservices should not be used for the sake of implementing a niche architecture style. It is extremely important to understand the business value and business KPIs before selecting microservices as an architectural solution for a given problem. A good understanding of business motivation and business value will help engineers focus on achieving these goals in a cost-effective way.

Business motivation and value should justify the selection of microservices. Also, using microservices, the business value should be realizable from a business point of view. This will avoid situations where IT invests in microservices but there is no appetite from the business to leverage any of the benefits that microservices can bring to the table. In such cases, a microservices-based development would be an overhead to the enterprise.

## Changing the mindset from project to product development

As discussed in *Chapter 1, Demystifying Microservices*, microservices are more aligned to product development. Business capabilities that are delivered using microservices should be treated as products. This is in line with the DevOps philosophy as well.

The mindset for project development and product development is different. The product team will always have a sense of ownership and take responsibility for what they produce. As a result, product teams always try to improve the quality of the product. The product team is responsible not only for delivering the software but also for production support and maintenance of the product.

Product teams are generally linked directly to a business department for which they are developing the product. In general, product teams have both an IT and a business representative. As a result, product thinking is closely aligned with actual business goals. At every moment, product teams understand the value they are adding to the business to achieve business goals. The success of the product directly lies with the business value being gained out of the product.

Because of the high-velocity release cycles, product teams always get a sense of satisfaction in their delivery, and they always try to improve on it. This brings a lot more positive dynamics within the team.

In many cases, typical product teams are funded for the long term and remain intact. As a result, product teams become more cohesive in nature. As they are small in size, such teams focus on improving their process from their day-to-day learnings.

One common pitfall in product development is that IT people represent the business in the product team. These IT representatives may not fully understand the business vision. Also, they may not be empowered to take decisions on behalf of the business. Such cases can result in a misalignment with the business and lead to failure quite rapidly.

It is also important to consider a collocation of teams where business and IT representatives reside at the same place. Collocation adds more binding between IT and business teams and reduces communication overheads.

## **Choosing a development philosophy**

Different organizations take different approaches to developing microservices, be it a migration or a new development. It is important to choose an approach that suits the organization. There is a wide verity of approaches available, out of which a few are explained in this section.

### **Design thinking**

Design thinking is an approach primarily used for innovation-centric development. It is an approach that explores the system from an end user point of view: what the customers see and how they experience the solution. A story is then built based on observations, patterns, intuition, and interviews.

Design thinking then quickly devises solutions through solution-focused thinking by employing a number of theories, logical reasoning, and assumptions around the problem. The concepts are expanded through brainstorming before arriving at a converged solution.

Once the solution is identified, a quick prototype is built to consider how the customer responds to it, and then the solution is adjusted accordingly. When the team gets satisfactory results, the next step is taken to scale the product. Note that the prototype may or may not be in the form of code.

Design thinking uses human-centric thinking with feelings, empathy, intuition, and imagination at its core. In this approach, solutions will be up for rethinking even for known problems to find innovative and better solutions

## **The start-up model**

More and more organizations are following the start-up philosophy to deliver solutions. Organizations create internal start-up teams with the mission to deliver specific solutions. Such teams stay away from day-to-day organizational activities and focus on delivering their mission.

Many start-ups kick off with a small, focused team—a highly cohesive unit. The unit is not worried about how they achieve things; rather, the focus is on what they want to achieve. Once they have a product in place, the team thinks about the right way to build and scale it.

This approach addresses quick delivery through production-first thinking. The advantage with this approach is that teams are not disturbed by organizational governance and political challenges. The team is empowered to think out of the box, be innovative, and deliver things. Generally, a higher level of ownership is seen in such teams, which is one of the key catalysts for success. Such teams employ just enough processes and disciplines to take the solution forward. They also follow a fail fast approach and course correct sooner than later.

## **The Agile practice**

The most commonly used approach is the Agile methodology for development. In this approach, software is delivered in an incremental, iterative way using the principles put forth in the Agile manifesto. This type of development uses an Agile method such as Scrum or XP. The Agile manifesto defines four key points that Agile software development teams should focus on:

- Individuals and interaction over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan



The 12 principles of Agile software development can be found at <http://www.agilemanifesto.org/principles.html>.

## Using the concept of Minimum Viable Product

Irrespective of the development philosophy explained earlier, it is essential to identify a **Minimum Viable Product (MVP)** when developing microservice systems for speed and agility.

Eric Ries, while pioneering the lean start-up movement, defined MVP as

*"A Minimum Viable Product is that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort."*

The objective of the MVP approach is to quickly build a piece of software that showcases the most important aspects of the software. The MVP approach realizes the core concept of an idea and perhaps chooses those features that add maximum value to the business. It helps get early feedback and then course corrects as necessary before building a heavy product.

The MVP may be a full-fledged service addressing limited user groups or partial services addressing wider user groups. Feedback from customers is extremely important in the MVP approach. Therefore, it is important to release the MVP to the real users.

## Overcoming the legacy hotspot

It is important to understand the environmental and political challenges in an organization before embarking on microservices development.

It is common in microservices to have dependencies on other legacy applications, directly or indirectly. A common issue with direct legacy integration is the slow development cycle of the legacy application. An example would be an innovative railway reservation system relaying on an age-old **transaction processing facility (TPF)** for some of the core backend features, such as reservation. This is especially common when migrating legacy monolithic applications to microservices. In many cases, legacy systems continue to undergo development in a non-Agile way with larger release cycles. In such cases, microservices development teams may not be able to move so quickly because of the coupling with legacy systems. Integration points might drag the microservices developments heavily. Organizational political challenges make things even worse.

There is no silver bullet to solve this issue. The cultural and process differences could be an ongoing issue. Many enterprises ring-fence such legacy systems with focused attention and investments to support fast-moving microservices. Targeted C-level interventions on these legacy platforms could reduce the overheads.

## Addressing challenges around databases

Automation is key in microservices development. Automating databases is one of the key challenges in many microservice developments.

In many organizations, DBAs play a critical role in database management, and they like to treat the databases under their control differently. Confidentiality and access control on data is also cited as a reason for DBAs to centrally manage all data.

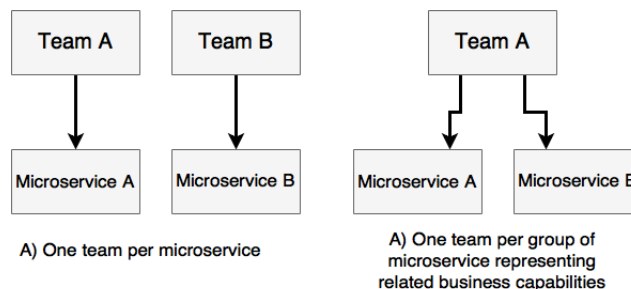
Many automation tools focus on the application logic. As a result, many development teams completely ignore database automation. Ignoring database automation can severely impact the overall benefits and can derail microservices development.

In order to avoid such situations, the database has to be treated in the same way as applications with appropriate source controls and change management. When selecting a database, it is also important to consider automation as one of the key aspects.

Database automation is much easier in the case of NoSQL databases but is hard to manage with traditional RDBMs. **Database Lifecycle Management (DLM)** as a concept is popular in the DevOps world, particularly to handle database automation. Tools such as DBmaestro, Redgate DLM, Datical DB, and Delphix support database automation.

## Establishing self-organizing teams

One of the most important activities in microservices development is to establish the right teams for development. As recommended in many DevOps processes, a small, focused team always delivers the best results.



As microservices are aligned with business capabilities and are fairly loosely coupled products, it is ideal to have a dedicated team per microservice. There could be cases where the same team owns multiple microservices from the same business area representing related capabilities. These are generally decided by the coupling and size of the microservices.

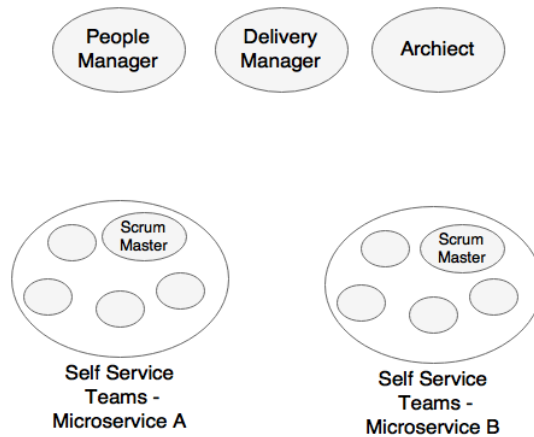
Team size is an important aspect in setting up effective teams for microservices development. The general notion is that the team size should not exceed 10 people. The recommended size for optimal delivery is between 4 and 7. The founder of Amazon.com, Jeff Bezos, coined the theory of two-pizza teams. Jeff's theory says the team will face communication issues if the size gets bigger. Larger teams work with consensus, which results in increased wastage. Large teams also lose ownership and accountability. A yardstick is that the product owner should get enough time to speak to individuals in the team to make them understand the value of what they are delivering.

Teams are expected to take full ownership in ideating for, analyzing, developing, and supporting services. Werner Vogels from Amazon.com calls this *you build it and you run it*. As per Werner's theory, developers pay more attention to develop quality code to avoid unexpected support calls. The members in the team consist of fullstack developers and operational engineers. Such a team is fully aware of all the areas. Developers understand operations as well as operations teams understand applications. This not only reduces the changes of throwing mud across teams but also improves quality.

Teams should have multidisciplinary skills to satisfy all the capabilities required to deliver a service. Ideally, the team should not rely on external teams to deliver the components of the service. Instead, the team should be self-sufficient. However, in most organizations, the challenge is on specialized skills that are rare. For example, there may not be many experts on a graph database in the organization. One common solution to this problem is to use the concept of consultants. Consultants are SMEs and are engaged to gain expertise on specific problems faced by the team. Some organizations also use shared or platform teams to deliver some common capabilities.

Team members should have a complete understanding of the products, not only from the technical standpoint but also from the business case and the business KPIs. The team should have collective ownership in delivering the product as well as in achieving business goals together.

Agile software development also encourages having self-organizing teams. Self-organizing teams act as a cohesive unit and find ways to achieve their goals as a team. The team automatically align themselves and distribute the responsibilities. The members in the team are self-managed and empowered to make decisions in their day-to-day work. The team's communication and transparency are extremely important in such teams. This emphasizes the need for collocation and collaboration, with a high bandwidth for communication:



In the preceding diagram, both **Microservice A** and **Microservice B** represent related business capabilities. Self-organizing teams treat everyone in the team equally, without too many hierarchies and management overheads within the team. The management would be thin in such cases. There won't be many designated vertical skills in the team, such as team lead, UX manager, development manager, testing manager, and so on. In a typical microservice development, a shared product manager, shared architect, and a shared people manager are good enough to manage the different microservice teams. In some organizations, architects also take up responsibility for delivery.

Self-organizing teams have some level of autonomy and are empowered to take decisions in a quick and Agile mode rather than having to wait for long-running bureaucratic decision-making processes that exist in many enterprises. In many of these cases, enterprise architecture and security are seen as an afterthought. However, it is important to have them on board from the beginning. While empowering the teams with maximum freedom for developers in decision-making capabilities, it is equally important to have fully automated QA and compliance so as to ensure that deviations are captured at the earliest.



Communication between teams is important. However, in an ideal world, it should be limited to interfaces between microservices. Integrations between teams ideally has to be handled through consumer-driven contracts in the form of test scripts rather than having large interface documents describing various scenarios. Teams should use mock service implementations when the services are not available.

## **Building a self-service cloud**

One of the key aspects that one should consider before embarking on microservices is to build a cloud environment. When there are only a few services, it is easy to manage them by manually assigning them to a certain predesignated set of virtual machines.

However, what microservice developers need is more than just an IaaS cloud platform. Neither the developers nor the operations engineers in the team should worry about where the application is deployed and how optimally it is deployed. They also should not worry about how the capacity is managed.

This level of sophistication requires a cloud platform with self-service capabilities, such as what we discussed in *Chapter 9, Managing Dockerized Microservices with Mesos and Marathon*, with the Mesos and Marathon cluster management solutions. Containerized deployment discussed in *Chapter 8, Containerizing Microservices with Docker*, is also important in managing and end to-end-automation. Building this self-service cloud ecosystem is a prerequisite for microservice development.

## **Building a microservices ecosystem**

As we discussed in the capability model in *Chapter 3, Applying Microservices Concepts*, microservices require a number of other capabilities. All these capabilities should be in place before implementing microservices at scale.

These capabilities include service registration, discovery, API gateways, and an externalized configuration service. All are provided by the Spring Cloud project. Capabilities such as centralized logging, monitoring, and so on are also required as a prerequisite for microservices development.

# Defining a DevOps-style microservice life cycle process

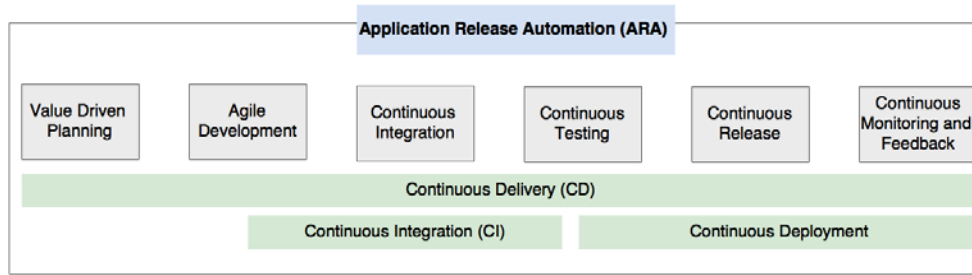
DevOps is the best-suited practice for microservices development. Organizations already practicing DevOps do not need another practice for microservices development.

In this section, we will explore the life cycle of microservices development. Rather than reinventing a process for microservices, we will explore DevOps processes and practices from the microservice perspective.

Before we explore DevOps processes, let's iron out some of the common terminologies used in the DevOps world:

- **Continuous integration (CI):** This automates the application build and quality checks continuously in a designated environment, either in a time-triggered manner or on developer commits. CI also publishes code metrics to a central dashboard as well as binary artifacts to a central repository. CI is popular in Agile development practices.
- **Continuous delivery (CD):** This automates the end-to-end software delivery practice from idea to production. In a non-DevOps model, this used to be known as **Application Lifecycle Management (ALM)**. One of the common interpretations of CD is that it is the next evolution of CI, which adds QA cycles into the integration pipeline and makes the software ready to release to production. A manual action is required to move it to production.
- **Continuous deployment:** This is an approach to automating the deployment of application binaries to one or more environments by managing binary movement and associated configuration parameters. Continuous deployment is also considered as the next evolution of CD by integrating automatic release processes into the CD pipeline.
- **Application Release Automation (ARA):** ARA tools help monitor and manage end-to-end delivery pipelines. ARA tools use CI and CD tools and manage the additional steps of release management approvals. ARA tools are also capable of rolling out releases to different environments and rolling them back in case of a failed deployment. ARA provides a fully orchestrated workflow pipeline, implementing delivery life cycles by integrating many specialized tools for repository management, quality assurance, deployment, and so on. XL Deploy and Automate are some of the ARA tools.

The following diagram shows the DevOps process for microservices development:



Let's now further explore these life cycle stages of microservices development.

## Value-driven planning

Value-driven planning is a term used in Agile development practices. Value-driven planning is extremely important in microservices development. In value-driven planning, we will identify which microservices to develop. The most important aspect is to identify those requirements that have the highest value to business and those that have the lowest risks. The MVP philosophy is used when developing microservices from the ground up. In the case of monolithic to microservices migration, we will use the guidelines provided in *Chapter 3, Applying Microservices Concepts*, to identify which services have to be taken first. The selected microservices are expected to precisely deliver the expected value to the business. Business KPIs to measure this value have to be identified as part of value-driven planning.

## Agile development

Once the microservices are identified, development must be carried out in an Agile approach following the Agile manifesto principles. The scrum methodology is used by most of the organizations for microservices development.

## Continuous integration

The continuous integration steps should be in place to automatically build the source code produced by various team members and generate binaries. It is important to build only once and then move the binary across the subsequent phases. Continuous integration also executes various QAs as part of the build pipeline, such as code coverage, security checks, design guidelines, and unit test cases. CI typically delivers binary artefacts to a binary artefact repository and also deploys the binary artefacts into one or more environments. Part of the functional testing also happens as part of CI.

## Continuous testing

Once continuous integration generates the binaries, they are moved to the testing phase. A fully automated testing cycle is kicked off in this phase. It is also important to automate security testing as part of the testing phase. Automated testing helps improve the quality of deliverables. The testing may happen in multiple environments based on the type of testing. This could range from the integration test environment to the production environment to test in production.

## Continuous release

Continuous release to production takes care of actual deployment, infrastructure provisioning, and rollout. The binaries are automatically shipped and deployed to production by applying certain rules. Many organizations stop automation with the staging environment and make use of manual approval steps to move to production.

## Continuous monitoring and feedback

The continuous monitoring and feedback phase is the most important phase in Agile microservices development. In an MVP scenario, this phase gives feedback on the initial acceptance of the MVP and also evaluates the value of the service developed. In a feature addition scenario, this further gives insight into how this new feature is accepted by users. Based on the feedback, the services are adjusted and the same cycle is then repeated.

## Automating the continuous delivery pipeline

In the previous section, we discussed the life cycle of microservices development. The life cycle stages can be altered by organizations based on their organizational needs but also based on the nature of the application. In this section, we will take a look at a sample continuous delivery pipeline as well as toolsets to implement a sample pipeline.

There are many tools available to build end-to-end pipelines, both in the open source and commercial space. Organizations can select the products of their choice to connect pipeline tasks.



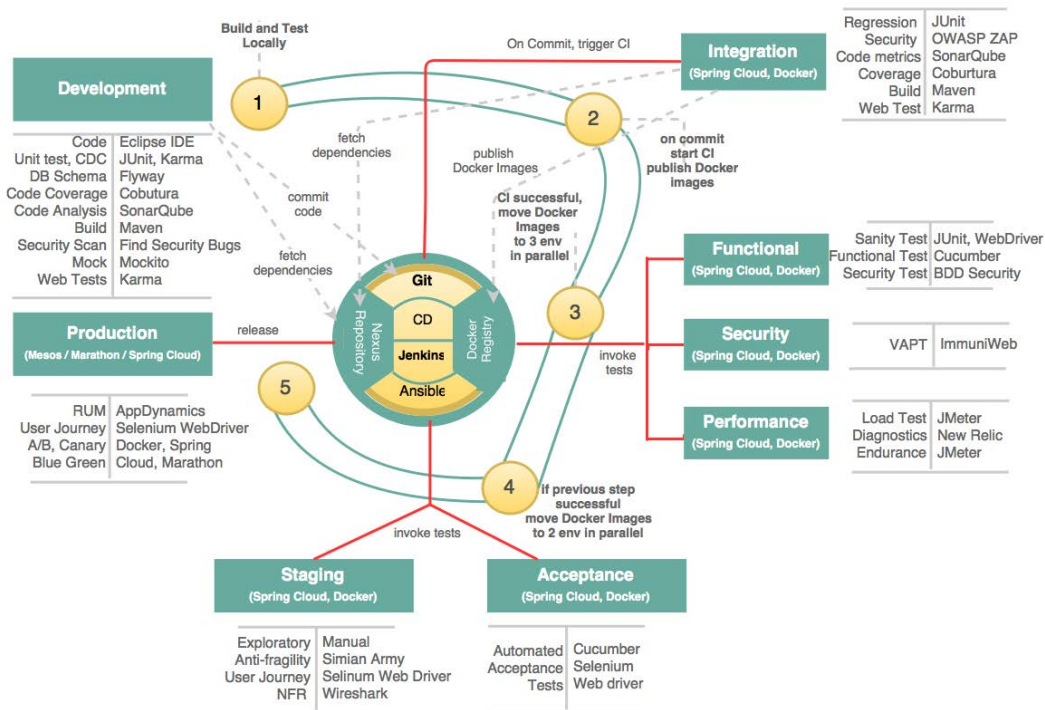
Refer to the XebiaLabs periodic table for a tool reference to build continuous delivery pipelines. It is available at <https://xebialabs.com/periodic-table-of-devops-tools/>.

The pipelines may initially be expensive to set up as they require many toolsets and environments. Organizations may not realize an immediate cost benefit in implementing the delivery pipeline. Also, building a pipeline needs high-power resources. Large build pipelines may involve hundreds of machines. It also takes hours to move changes through the pipeline from one end to the other. Hence, it is important to have different pipelines for different microservices. This will also help decoupling between the releases of different microservices.

Within a pipeline, parallelism should be employed to execute tests on different environments. It is also important to parallelize the execution of test cases as much as possible. Hence, designing the pipeline based on the nature of the application is important. There is no one size fits all scenario

The key focus in the pipeline is on end-to-end automation, from development to production, and on failing fast if something goes wrong.

The following pipeline is an indicative one for microservices and explores the different capabilities that one should consider when developing a microservices pipeline:



The continuous delivery pipeline stages are explained in the following sections.

## Development

The development stage has the following activities from a development perspective. This section also indicates some of the tools that can be used in the development stage. These tools are in addition to the planning, tracking, and communication tools such as Agile JIRA, Slack, and others used by Agile development teams. Take a look at the following:

- **Source code:** The development team requires an IDE or a development environment to cut source code. In most organizations, developers get the freedom to choose the IDEs they want. Having said this, the IDEs can be integrated with a number of tools to detect violations against guidelines. Generally, Eclipse IDEs have plugins for static code analysis and code matrices. SonarQube is one example that integrates other plugins such as Checkstyle for code conventions, PMD to detect bad practices, FindBugs to detect potential bugs, and Cobertura for code coverage. It is also recommended to use Eclipse plugins such as ESVD, Find Security Bugs, SonarQube Security Rules, and so on to detect security vulnerabilities.
- **Unit test cases:** The development team also produces unit test cases using JUnit, NUnit, TestNG, and so on. Unit test cases are written against components, repositories, services, and so on. These unit test cases are integrated with the local Maven builds. The unit test cases targeting the microservice endpoints (service tests) serve as the regression test pack. Web UI, if written in AngularJS, can be tested using Karma.
- **Consumer-driven contracts:** Developers also write CDCs to test integration points with other microservices. Contract test cases are generally written as JUnit, NUnit, TestNG, and so on and are added to the service tests pack mentioned in the earlier steps.
- **Mock testing:** Developers also write mocks to simulate the integration endpoints to execute unit test cases. Mockito, PowerMock, and others are generally used for mock testing. It is good practice to deploy a mock service based on the contract as soon as the service contract is identified. This acts as a simple mechanism for service virtualization for the subsequent phases.
- **Behavior driven design (BDD):** The Agile team also writes BDD scenarios using a BDD tool, such as Cucumber. Typically, these scenarios are targeted against the microservices contract or the user interface that is exposed by a microservice-based web application. Cucumber with JUnit and Cucumber with Selenium WebDriver, respectively, are used in these scenarios. Different scenarios are used for functional testing, user journey testing, as well as acceptance testing.

- **Source code repository:** A source control repository is a part and parcel of development. Developers check-in their code to a central repository, mostly with the help of IDE plugins. One microservice per repository is a common pattern used by many organizations. This disallows other microservice developers from modifying other microservices or writing code based on the internal representations of other microservices. Git and Subversion are the popular choices to be used as source code repositories.
- **Build tools:** A build tool such as Maven or Gradle is used to manage dependencies and build target artifacts – in this case, Spring Boot services. There are many cases, such as basic quality checks, security checks and unit test cases, code coverage, and so on, that are integrated as part of the build itself. These are similar to the IDE, especially when IDEs are not used by developers. The tools that we examined as part of the IDEs are also available as Maven plugins. The development team does not use containers such as Docker until the CI phase of the project. All the artifacts have to be versioned properly for every change.
- **Artifact repository:** The artifact repository plays a pivotal role in the development process. The artifact repository is where all build artifacts are stored. The artifact repository could be Artifactory, Nexus, or any similar product.
- **Database schemas:** Liquibase and Flyway are commonly used to manage, track, and apply database changes. Maven plugins allow interaction with the Liquibase or Flyway libraries. The schema changes are versioned and maintained, just like source code.

## Continuous integration

Once the code is committed to the repository, the next phase, continuous integration, automatically starts. This is done by configuring a CI pipeline. This phase builds the source code with a repository snapshot and generates deployable artifacts. Different organizations use different events to kickstart the build. A CI start event may be on every developer commit or may be based on a time window, such as daily, weekly, and so on.

The CI workflow is the key aspect of this phase. Continuous integration tools such as Jenkins, Bamboo, and others play the central role of orchestrating the build pipeline. The tool is configured with a workflow of activities to be invoked. The workflow automatically executes configured steps such as build, deploy, and QA. On the developer commit or on a set frequency, the CI kickstarts the workflow

The following activities take place in a continuous integration workflow

1. **Build and QA:** The workflow listens to Git webhooks for commits. Once it detects a change, the first activity is to download the source code from the repository. A build is executed on the downloaded snapshot source code. As part of the build, a number of QA checks are automatically performed, similarly to QA executed in the development environment. These include code quality checks, security checks, and code coverage. Many of the QAs are done with tools such as SonarQube, with the plugins mentioned earlier. It also collects code metrics such as code coverage and more and publishes it to a central database for analysis. Additional security checks are executed using OWASP ZAP Jenkins' plugins. As part of the build, it also executes JUnit or similar tools used to write test cases. If the web application supports Karma for UI testing, Jenkins is also capable of running web tests written in Karma. If the build or QA fails, it sends out alarms as configured in the system
2. **Packaging:** Once build and QA are passed, the CI creates a deployable package. In our microservices case, it generates the Spring Boot standalone JAR. It is recommended to build Docker images as part of the integration build. This is the one and only place where we build binary artifacts. Once the build is complete, it pushes the immutable Docker images to a Docker registry. This could be on Docker Hub or a private Docker registry. It is important to properly version control the containers at this stage itself.
3. **Integration tests:** The Docker image is moved to the integration environment where regression tests (service tests) and the like are executed. This environment has other dependent microservices capabilities, such as Spring Cloud, logging, and so on, in place. All dependent microservices are also present in this environment. If an actual dependent service is not yet deployed, service virtualization tools such as MockServer are used. Alternately, a base version of the service is pushed to Git by the respective development teams. Once successfully deployed, Jenkins triggers service tests (JUnits against services), a set of end-to-end sanity tests written in Selenium WebDriver (in the case of web) and security tests with OWASP ZAP.

## Automated testing

There are many types of testing to be executed as part of the automated delivery process before declaring the build ready for production. The testing may happen by moving the application across multiple environments. Each environment is designated for a particular kind of testing, such as acceptance testing, performance testing, and so on. These environments are adequately monitored to gather the respective metrics.



In a complex microservices environment, testing should not be seen as a last-minute gate check; rather, testing should be considered as a way to improve software quality as well as to avoid last-minute failures. Shift left testing is an approach of shifting tests as early as possible in the release cycle. Automated testing turns software development to every-day development and every-day testing mode. By automating test cases, we will avoid manual errors as well as the effort required to complete testing.

CI or ARA tools are used to move Docker images across multiple test environments. Once deployed in an environment, test cases are executed based on the purpose of the environment. By default, a set of sanity tests are executed to verify the test environment.

In this section, we will cover all the types of tests that are required in the automated delivery pipeline, irrespective of the environment. We have already considered some types of tests as part of the development and integration environment. Later in this section, we will also map test cases against the environments in which they are executed.

## **Different candidate tests for automation**

In this section, we will explore different types of tests that are candidates for automation when designing an end-to-end delivery pipeline. The key testing types are described as follows.

### **Automated sanity tests**

When moving from one environment to another, it is advisable to run a few sanity tests to make sure that all the basic things are working. This is created as a test pack using JUnit service tests, Selenium WebDriver, or a similar tool. It is important to carefully identify and script all the critical service calls. Especially if the microservices are integrated using synchronous dependencies, it is better to consider these scenarios to ensure that all dependent services are also up and running.

### **Regression testing**

Regression tests ensure that changes in software don't break the system. In a microservices context, the regression tests could be at the service level (Rest API or message endpoints) and written using JUnit or a similar framework, as explained earlier. Service virtualizations are used when dependent services are not available. Karma and Jasmine can be used for web UI testing.

In cases where microservices are used behind web applications, Selenium WebDriver or a similar tool is used to prepare regression test packs, and tests are conducted at the UI level rather than focusing on the service endpoints. Alternatively, BDD tools, such as Cucumber with JUnit or Cucumber with Selenium WebDriver, can also be used to prepare regression test packs. CI tools such as Jenkins or ARA are used to automatically trigger regression test packs. There are other commercial tools, such as TestComplete, that can also be used to build regression test packs.

## Automated functional testing

Functional test cases are generally targeted at the UIs that consume the microservices. These are business scenarios based on user stories or features. These functional tests are executed on every build to ensure that the microservice is performing as expected.

BDD is generally used in developing functional test cases. Typically in BDD, business analysts write test cases in a domain-specific language but in plain English. Developers then add scripts to execute these scenarios. Automated web testing tools such as Selenium WebDriver are useful in such scenarios, together with BDD tools such as Cucumber, JBehave, SpecFlow, and so on. JUnit test cases are used in the case of headless microservices. There are pipelines that combine both regression testing and functional testing as one step with the same set of test cases.

## Automated acceptance testing

This is much similar to the preceding functional test cases. In many cases, automated acceptance tests generally use the screenplay or journey pattern and are applied at the web application level. The customer perspective is used in building the test cases rather than features or functions. These tests mimic user flows

BDD tools such as Cucumber, JBehave, and SpecFlow are generally used in these scenarios together with JUnit or Selenium WebDriver, as discussed in the previous scenario. The nature of the test cases is different in functional testing and acceptance testing. Automation of acceptance test packs is achieved by integrating them with Jenkins. There are many other specialized automatic acceptance testing tools available on the market. FitNesse is one such tool.

## Performance testing

It is important to automate performance testing as part of the delivery pipeline. This positions performance testing from a gate check model to an integral part of the delivery pipeline. By doing so, bottlenecks can be identified at very early stages of build cycles. In some organizations, performance tests are conducted only for major releases, but in others, performance tests are part of the pipeline. There are multiple options for performance testing. Tools such as JMeter, Gatling, Grinder, and so on can be used for load testing. These tools can be integrated into the Jenkins workflow for automation. Tools such as BlazeMeter can then be used for test reporting.

Application Performance Management tools such as AppDynamics, New Relic, Dynatrace, and so on provide quality metrics as part of the delivery pipeline. This can be done using these tools as part of the performance testing environment. In some pipelines, these are integrated into the functional testing environment to get better coverage. Jenkins has plugins in to fetch measurements.

## Real user flow simulation or journey testing

This is another form of test typically used in staging and production environments. These tests continuously run in staging and production environments to ensure that all the critical transactions perform as expected. This is much more useful than a typical URL ping monitoring mechanism. Generally, similar to automated acceptance testing, these test cases simulate user journeys as they happen in the real world. These are also useful to check whether the dependent microservices are up and running. These test cases could be a carved-out subset of acceptance test cases or test packs created using Selenium WebDriver.

## Automated security testing

It is extremely important to make sure that the automation does not violate the security policies of the organization. Security is the most important thing, and compromising security for speed is not desirable. Hence, it is important to integrate security testing as part of the delivery pipeline. Some security evaluations are already integrated in the local build environment as well as in the integration environment, such as SonarQube, Find Security Bugs, and so on. Some security aspects are covered as part of the functional test cases. Tools such as BDD-Security, Mittn, and Gauntlt are other security test automation tools following the BDD approach. VAPT can be done using tools such as ImmuniWeb. OWASP ZAP and Burp Suite are other useful tools in security testing.

## Exploratory testing

Exploratory testing is a manual testing approach taken by testers or business users to validate the specific scenarios that they think automated tools may not capture. Testers interact with the system in any manner they want without prejudice. They use their intellect to identify the scenarios that they think some special users may explore. They also do exploratory testing by simulating certain user behavior.

## A/B testing, canary testing, and blue-green deployments

When moving applications to production, A/B testing, blue-green deployments, and canary testing are generally applied. A/B testing is primarily used to review the effectiveness of a change and how the market reacts to the change. New features are rolled out to a certain set of users. Canary release is moving a new product or feature to a certain community before fully rolling out to all customers. Blue-green is a deployment strategy from an IT point of view to test the new version of a service. In this model, both blue and green versions are up and running at some point of time and then gracefully migrate from one to the other.

## Other nonfunctional tests

High availability and antifragility testing (failure injection tests) are also important to execute before production. This helps developers unearth unknown errors that may occur in a real production scenario. This is generally done by breaking the components of the system to understand their failover behavior. This is also helpful to test circuit breakers and fallback services in the system. Tools such as Simian Army are useful in these scenarios.

## Testing in production

**Testing in Production (TiP)** is as important as all the other environments as we can only simulate to a certain extent. There are two types of tests generally executed against production. The first approach is running real user flows or journey tests in continuous manner, simulating various user actions. This is automated using one of the **Real User Monitoring (RUM)** tools, such as AppDynamics. The second approach is to wiretap messages from production, execute them in a staging environment, and then compare the results in production with those in the staging environment.

## Antifragility testing

Antifragility testing is generally conducted in a preproduction environment identical to production or even in the production environment by creating chaos in the environment to take a look at how the application responds and recovers from these situations. Over a period of time, the application gains the ability to automatically recover from most of these failures. Simian Army is one such tool from Netflix. Simian Army is a suite of products built for the AWS environment. Simian Army is for disruptive testing using a set of autonomous monkeys that can create chaos in the preproduction or production environments. Chaos Monkey, Janitor Monkey, and Conformity Monkey are some of the components of Simian Army.

## Target test environments

The different test environments and the types of tests targeted on these environments for execution are as follows:

- **Development environment:** The development environment is used to test the coding style checks, bad practices, potential bugs, unit tests, and basic security scanning.
- **Integration test environment:** Integration environment is used for unit testing and regression tests that span across multiple microservices. Some basic security-related tests are also executed in the integration test environment.
- **Performance and diagnostics:** Performance tests are executed in the performance test environment. Application performance testing tools are deployed in these environments to collect performance metrics and identify bottlenecks.
- **Functional test environment:** The functional test environment is used to execute a sanity test and functional test packs.
- **UAT environment:** The UAT environment has sanity tests, automated acceptance test packs, and user journey simulations.
- **Staging:** The preproduction environment is used primarily for sanity tests, security, antifragility, network tests, and so on. It is also used for user journey simulations and exploratory testing.
- **Production:** User journey simulations and RUM tests are continuously executed in the production environment.

Making proper data available across multiple environments to support test cases is the biggest challenge. Delphix is a useful tool to consider when dealing with test data across multiple environments in an effective way.

## Continuous deployment

Continuous deployment is the process of deploying applications to one or more environments and configuring and provisioning these environments accordingly. As discussed in *Chapter 9, Managing Dockerized Microservices with Mesos and Marathon*, infrastructure provisioning and automation tools facilitate deployment automation.

From the deployment perspective, the released Docker images are moved to production automatically once all the quality checks are successfully completed. The production environment, in this case, has to be cloud based with a cluster management tool such as Mesos or Marathon. A self-service cloud environment with monitoring capabilities is mandatory.

Cluster management and application deployment tools ensure that application dependencies are properly deployed. This automatically deploys all the dependencies that are required in case any are missing. It also ensures that a minimum number of instances are running at any point in time. In case of failure, it automatically rolls back the deployments. It also takes care of rolling back upgrades in a graceful manner.

Ansible, Chef, or Puppet are tools useful in moving configurations and binaries to production. The Ansible playbook concepts can be used to launch a Mesos cluster with Marathon and Docker support.

## Monitoring and feedback

Once an application is deployed in production, monitoring tools continuously monitor its services. Monitoring and log management tools collect and analyze information. Based on the feedback and corrective actions needed, information is fed to the development teams to take corrective actions, and the changes are pushed back to production through the pipeline. Tools such as APM, Open Web Analytics, Google Analytics, Webalizer, and so on are useful tools to monitor web applications. Real user monitoring should provide end-to-end monitoring. QuBit, Boxever, Channel Site, MaxTraffic, and so on are also useful in analyzing customer behavior

## Automated configuration management

Configuration management also has to be rethought from a microservices and DevOps perspective. Use new methods for configuration management rather than using a traditional statically configured CMDB. The manual maintenance of CMDB is no longer an option. Statically managed CMDB requires a lot of mundane tasks to maintain entries. At the same time, due to the dynamic nature of the deployment topology, it is extremely hard to maintain data in a consistent way.

The new styles of CMDB automatically create CI configurations based on an operational topology. These should be discovery based to get up-to-date information. The new CMDB should be capable of managing bare metals, virtual machines, and containers.

## **Microservices development governance, reference architectures, and libraries**

It is important to have an overall enterprise reference architecture and a standard set of tools for microservices development to ensure that development is done in a consistent manner. This helps individual microservices teams to adhere to certain best practices. Each team may identify specialized technologies and tools that are suitable for their development. In a polyglot microservices development, there are obviously multiple technologies used by different teams. However, they have to adhere to the arching principles and practices.

For quick wins and to take advantage of timelines, microservices development teams may deviate from these practices in some cases. This is acceptable as long as the teams add refactoring tasks in their backlogs. In many organizations, although the teams make attempts to reuse something from the enterprise, reuse and standardization generally come as an afterthought.

It is important to make sure that the services are catalogued and visible in the enterprise. This improves the reuse opportunities of microservices.

## **Summary**

In this chapter, you learned about the relationship between microservices and DevOps. We also examined a number of practice points when developing microservices. Most importantly, you learned the microservices development life cycle.

Later in this chapter, we also examined how to automate the microservices delivery pipeline from development to production. As part of this, we examined a number of tools and technologies that are helpful when automating the microservices delivery pipeline. Finally, we touched base with the importance of reference architectures in microservices governance.

Putting together the concepts of microservices, challenges, best practices, and various capabilities covered in this book makes a perfect recipe for developing successful microservices at scale.

# Bibliography

This learning path has been prepared for you to explore Spring's features and learn to use them in your existing projects. It comprises of the following Packt products:

- *Learning Spring Application Development* by Ravi Kant Soni
- *Spring MVC Beginner's Guide - Second Edition* by Amuthan Ganeshan
- *Spring Microservices* by Rajesh RV







# Thank you for buying **Spring: Developing Java Applications for the Enterprise**

## About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at [www.packtpub.com](http://www.packtpub.com).

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

